

(old)

htmldiff from-

(new)

## Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or TMTM are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

Copyright © 2018 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:18-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

(old)

htmldiff from-

(new)

## A64 -- Base Instructions (alphabetic order)

ADC: Add with Carry.

ADCS: Add with Carry, setting flags.

[ADD \(extended register\)](#): Add (extended register).

[ADD \(immediate\)](#): Add (immediate).

[ADD \(shifted register\)](#): Add (shifted register).

[ADDG](#): Add with Tag.

[ADDS \(extended register\)](#): Add (extended register), setting flags.

[ADDS \(immediate\)](#): Add (immediate), setting flags.

[ADDS \(shifted register\)](#): Add (shifted register), setting flags.

ADR: Form PC-relative address.

ADRP: Form PC-relative address to 4KB page.

[AND \(immediate\)](#): Bitwise AND (immediate).

[AND \(shifted register\)](#): Bitwise AND (shifted register).

[ANDS \(immediate\)](#): Bitwise AND (immediate), setting flags.

[ANDS \(shifted register\)](#): Bitwise AND (shifted register), setting flags.

ASR (immediate): Arithmetic Shift Right (immediate): an alias of SBFM.

ASR (register): Arithmetic Shift Right (register): an alias of ASRV.

ASRV: Arithmetic Shift Right Variable.

AT: Address Translate: an alias of SYS.

[AUTDA, AUTDZA](#): Authenticate Data address, using key A.

[AUTDB, AUTDZB](#): Authenticate Data address, using key B.

[AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA](#): Authenticate Instruction address, using key A.

[AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB](#): Authenticate Instruction address, using key B.

[AXFlag](#): Convert floating-point condition flags from ARM to external format.

[B](#): Branch.

[B.cond](#): Branch conditionally.

BFC: Bitfield Clear: an alias of BFM.

BFI: Bitfield Insert: an alias of BFM.

[BFM](#): Bitfield Move.

BFXIL: Bitfield extract and insert at low end: an alias of BFM.

[BIC \(shifted register\)](#): Bitwise Bit Clear (shifted register).

[BICS \(shifted register\)](#): Bitwise Bit Clear (shifted register), setting flags.

[BL](#): Branch with Link.

[BLR](#): Branch with Link to Register.

[BLRAA](#), [BLRAAZ](#), [BLRAB](#), [BLRABZ](#): Branch with Link to Register, with pointer authentication.

[BR](#): Branch to Register.

[BRAA](#), [BRAAZ](#), [BRAB](#), [BRABZ](#): Branch to Register, with pointer authentication.

[BRK](#): Breakpoint instruction.

[BTI](#): Branch Target Identification.

[CAS](#), [CASA](#), [CASAL](#), [CASL](#): Compare and Swap word or doubleword in memory.

[CASB](#), [CASAB](#), [CASALB](#), [CASLB](#): Compare and Swap byte in memory.

[CASH](#), [CASHA](#), [CASALH](#), [CASLH](#): Compare and Swap halfword in memory.

[CASP](#), [CASPA](#), [CASPAL](#), [CASPL](#): Compare and Swap Pair of words or doublewords in memory.

[CBNZ](#): Compare and Branch on Nonzero.

[CBZ](#): Compare and Branch on Zero.

CCMN (immediate): Conditional Compare Negative (immediate).

CCMN (register): Conditional Compare Negative (register).

CCMP (immediate): Conditional Compare (immediate).

CCMP (register): Conditional Compare (register).

[CFINV](#): ~~Invert~~ ~~PSTATE.C~~ ~~Carry Flag~~ ~~Flag Inversion~~.

[CFP](#): Control Flow Prediction Restriction by Context: an alias of SYS.

CINC: Conditional Increment: an alias of CSINC.

CINV: Conditional Invert: an alias of CSINV.

CLREX: Clear Exclusive.

[CLS](#): Count ~~Leading~~ ~~leading~~ ~~Sign~~ ~~sign~~ bits.

[CLZ](#): Count ~~Leading~~ ~~leading~~ ~~Zeros~~ ~~zero~~ bits.

CMN (extended register): Compare Negative (extended register): an alias of ADDS (extended register).

[CMN \(immediate\)](#): Compare Negative (immediate): an alias of ADDS (immediate).

CMN (shifted register): Compare Negative (shifted register): an alias of ADDS (shifted register).

CMP (extended register): Compare (extended register): an alias of SUBS (extended register).

[CMP \(immediate\)](#): Compare (immediate): an alias of SUBS (immediate).

CMP (shifted register): Compare (shifted register): an alias of SUBS (shifted register).

[CMPP](#): Compare with Tag: an alias of SUBPS.

CNEG: Conditional Negate: an alias of CSNEG.

[CPP](#): Cache Prefetch Prediction Restriction by Context: an alias of SYS.

[CRC32B](#), [CRC32H](#), [CRC32W](#), [CRC32X](#): CRC32 checksum.

[CRC32CB](#), [CRC32CH](#), [CRC32CW](#), [CRC32CX](#): CRC32C checksum.

[CSDB](#): Consumption of Speculative Data Barrier.

CSEL: Conditional Select.

**CSET**: Conditional Set: an alias of CSINC.

**CSETM**: Conditional Set Mask: an alias of CSINV.

**CSINC**: Conditional Select Increment.

**CSINV**: Conditional Select Invert.

**CSNEG**: Conditional Select Negation.

**DC**: Data Cache operation: an alias of SYS.

**DCPS1**: Debug Change PE State to EL1..

**DCPS2**: Debug Change PE State to EL2..

**DCPS3**: Debug Change PE State to EL3.

**DMB**: Data Memory Barrier.

**DRPS**: Debug restore process state.

**DSB**: Data Synchronization Barrier.

**DVP**: Data Value Prediction Restriction by Context: an alias of SYS.

**EON (shifted register)**: Bitwise Exclusive OR NOT (shifted register).

**EOR (immediate)**: Bitwise Exclusive OR (immediate).

**EOR (shifted register)**: Bitwise Exclusive OR (shifted register).

**ERET**: Exception Return.

**ERETAA, ERETAB**: Exception Return, with pointer authentication.

**ESB**: Error Synchronization Barrier.

**EXTR**: Extract register.

**GMI**: Tag Mask Insert.

**HINT**: Hint instruction.

**HLT**: Halt instruction.

**HVC**: Hypervisor Call.

**IC**: Instruction Cache operation: an alias of SYS.

**IRG**: Insert Random Tag.

**ISB**: Instruction Synchronization Barrier.

**LDADD, LDADDA, LDADDAL, LDADDL**: Atomic add on word or doubleword in memory.

**LDADDB, LDADDAB, LDADDALB, LDADDLB**: Atomic add on byte in memory.

**LDADDH, LDADDAH, LDADDALH, LDADDLH**: Atomic add on halfword in memory.

**LDAPR**: Load-Acquire RCpc Register.

**LDAPRB**: Load-Acquire RCpc Register Byte.

**LDAPRH**: Load-Acquire RCpc Register Halfword.

**LDAPUR**: Load-Acquire RCpc Register (unscaled).

**LDAPURB**: Load-Acquire RCpc Register Byte (unscaled).

**LDAPURH**: Load-Acquire RCpc Register Halfword (unscaled).



[LDAPURSB](#): Load-Acquire RCpc Register Signed Byte (unscaled).

[LDAPURSH](#): Load-Acquire RCpc Register Signed Halfword (unscaled).

[LDAPURSW](#): Load-Acquire RCpc Register Signed Word (unscaled).

[LDAR](#): Load-Acquire Register.

[LDARB](#): Load-Acquire Register Byte.

[LDARH](#): Load-Acquire Register Halfword.

[LDAXP](#): Load-Acquire Exclusive Pair of Registers.

[LDAXR](#): Load-Acquire Exclusive Register.

[LDAXRB](#): Load-Acquire Exclusive Register Byte.

[LDAXRH](#): Load-Acquire Exclusive Register Halfword.

[LDCLR](#), [LDCLRA](#), [LDCLRAL](#), [LDCLRL](#): Atomic bit clear on word or doubleword in memory.

[LDCLRB](#), [LDCLRAB](#), [LDCLRALB](#), [LDCLRLB](#): Atomic bit clear on byte in memory.

[LDCLRH](#), [LDCLRAH](#), [LDCLRALH](#), [LDCLRLH](#): Atomic bit clear on halfword in memory.

[LDEOR](#), [LDEORA](#), [LDEORAL](#), [LDEORL](#): Atomic exclusive OR on word or doubleword in memory.

[LDEORB](#), [LDEORAB](#), [LDEORALB](#), [LDEORLB](#): Atomic exclusive OR on byte in memory.

[LDEORH](#), [LDEORAH](#), [LDEORALH](#), [LDEORLH](#): Atomic exclusive OR on halfword in memory.

[LDG](#): Load Allocation Tag.

[LDGV](#): Load Allocation Tag.

[LDLAR](#): Load LOAcquire Register.

[LDLARB](#): Load LOAcquire Register Byte.

[LDLARH](#): Load LOAcquire Register Halfword.

[LDNP](#): Load Pair of Registers, with non-temporal hint.

[LDP](#): Load Pair of Registers.

[LDPSW](#): Load Pair of Registers Signed Word.

[LDR \(immediate\)](#): Load Register (immediate).

[LDR \(literal\)](#): Load Register (literal).

[LDR \(register\)](#): Load Register (register).

[LDRAA](#), [LDRAAB](#): Load Register, with pointer authentication.

[LDRB \(immediate\)](#): Load Register Byte (immediate).

[LDRB \(register\)](#): Load Register Byte (register).

[LDRH \(immediate\)](#): Load Register Halfword (immediate).

[LDRH \(register\)](#): Load Register Halfword (register).

[LDRSB \(immediate\)](#): Load Register Signed Byte (immediate).

[LDRSB \(register\)](#): Load Register Signed Byte (register).

[LDRSH \(immediate\)](#): Load Register Signed Halfword (immediate).

[LDRSH \(register\)](#): Load Register Signed Halfword (register).

[LDRSW \(immediate\)](#): Load Register Signed Word (immediate).

LDRSW (literal): Load Register Signed Word (literal).

[LDRSW \(register\)](#): Load Register Signed Word (register).

[LDSET, LDSETA, LDSETAL, LDSETL](#): Atomic bit set on word or doubleword in memory.

[LDSETB, LDSETAB, LDSETALB, LDSETLB](#): Atomic bit set on byte in memory.

[LDSETH, LDSETAH, LDSETALH, LDSETLH](#): Atomic bit set on halfword in memory.

[LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL](#): Atomic signed maximum on word or doubleword in memory.

[LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB](#): Atomic signed maximum on byte in memory.

[LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH](#): Atomic signed maximum on halfword in memory.

[LDSMIN, LDSMINA, LDSMINAL, LDSMINL](#): Atomic signed minimum on word or doubleword in memory.

[LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB](#): Atomic signed minimum on byte in memory.

[LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH](#): Atomic signed minimum on halfword in memory.

[LDTR](#): Load Register (unprivileged).

[LDTRB](#): Load Register Byte (unprivileged).

[LDTRH](#): Load Register Halfword (unprivileged).

[LDTRSB](#): Load Register Signed Byte (unprivileged).

[LDTRSH](#): Load Register Signed Halfword (unprivileged).

[LDTRSW](#): Load Register Signed Word (unprivileged).

[LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL](#): Atomic unsigned maximum on word or doubleword in memory.

[LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB](#): Atomic unsigned maximum on byte in memory.

[LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH](#): Atomic unsigned maximum on halfword in memory.

[LDUMIN, LDUMINA, LDUMINAL, LDUMINL](#): Atomic unsigned minimum on word or doubleword in memory.

[LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB](#): Atomic unsigned minimum on byte in memory.

[LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH](#): Atomic unsigned minimum on halfword in memory.

[LDUR](#): Load Register (unscaled).

[LDURB](#): Load Register Byte (unscaled).

[LDURH](#): Load Register Halfword (unscaled).

[LDURSB](#): Load Register Signed Byte (unscaled).

[LDURSH](#): Load Register Signed Halfword (unscaled).

[LDURSW](#): Load Register Signed Word (unscaled).

[LDXP](#): Load Exclusive Pair of Registers.

[LDXR](#): Load Exclusive Register.

[LDXRB](#): Load Exclusive Register Byte.

[LDXRH](#): Load Exclusive Register Halfword.

LSL (immediate): Logical Shift Left (immediate): an alias of UBFM.

LSL (register): Logical Shift Left (register): an alias of LSLV.

LSLV: Logical Shift Left Variable.

LSR (immediate): Logical Shift Right (immediate): an alias of UBFM.

LSR (register): Logical Shift Right (register): an alias of LSRV.

LSRV: Logical Shift Right Variable.

MADD: Multiply-Add.

MNEG: Multiply-Negate: an alias of MSUB.

MOV (bitmask immediate): Move (bitmask immediate): an alias of ORR (immediate).

MOV (inverted wide immediate): Move (inverted wide immediate): an alias of MOVN.

MOV (register): Move (register): an alias of ORR (shifted register).

[MOV \(to/from SP\)](#): Move between register and stack pointer: an alias of ADD (immediate).

MOV (wide immediate): Move (wide immediate): an alias of MOVZ.

[MOVK](#): Move wide with keep.

[MOVN](#): Move wide with NOT.

[MOVZ](#): Move wide with zero.

MRS: Move System Register.

[MSR \(immediate\)](#): Move immediate value to Special Register.

MSR (register): Move general-purpose register to System Register.

MSUB: Multiply-Subtract.

MUL: Multiply: an alias of MADD.

MVN: Bitwise NOT: an alias of ORN (shifted register).

NEG (shifted register): Negate (shifted register): an alias of SUB (shifted register).

NEGS: Negate, setting flags: an alias of SUBS (shifted register).

NGC: Negate with Carry: an alias of SBC.

NGCS: Negate with Carry, setting flags: an alias of SBCS.

[NOP](#): No Operation.

[ORN \(shifted register\)](#): Bitwise OR NOT (shifted register).

[ORR \(immediate\)](#): Bitwise OR (immediate).

[ORR \(shifted register\)](#): Bitwise OR (shifted register).

[PACDA, PACDZA](#): Pointer Authentication Code for Data address, using key A.

[PACDB, PACDZB](#): Pointer Authentication Code for Data address, using key B.

[PACGA](#): Pointer Authentication Code, using Generic key.

[PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA](#): Pointer Authentication Code for Instruction address, using key A.

[PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB](#): Pointer Authentication Code for Instruction address, using key B.

[PRFM \(immediate\)](#): Prefetch Memory (immediate).

PRFM (literal): Prefetch Memory (literal).

[PRFM \(register\)](#): Prefetch Memory (register).

[PRFM \(unscaled offset\)](#): Prefetch Memory (unscaled offset).

[PSB CSYNC](#): Profiling Synchronization Barrier.

[PSSBB](#): Physical Speculative Store Bypass Barrier.

RBIT: Reverse Bits.

[RET](#): Return from subroutine.

[RETAA, RETAB](#): Return from subroutine, with pointer authentication.

[REV](#): Reverse Bytes.

[REV16](#): Reverse bytes in 16-bit halfwords.

[REV32](#): Reverse bytes in 32-bit words.

[REV64](#): Reverse Bytes: an alias of REV.

[RMIF](#): Rotate, Mask Insert Flags.

ROR (immediate): Rotate right (immediate): an alias of EXTR.

ROR (register): Rotate Right (register): an alias of RORV.

RORV: Rotate Right Variable.

[SB](#): Speculation Barrier.

SBC: Subtract with Carry.

SBCS: Subtract with Carry, setting flags.

SBFIZ: Signed Bitfield Insert in Zero: an alias of SBFM.

[SBFM](#): Signed Bitfield Move.

SBFX: Signed Bitfield Extract: an alias of SBFM.

SDIV: Signed Divide.

[SETF8, SETF16](#): Evaluation of 8 or 16 bit flag values.

[SEV](#): Send Event.

[SEVL](#): Send Event Local.

SMADDL: Signed Multiply-Add Long.

SMC: Secure Monitor Call.

SMNEGL: Signed Multiply-Negate Long: an alias of SMSUBL.

SMSUBL: Signed Multiply-Subtract Long.

SMULH: Signed Multiply High.

SMULL: Signed Multiply Long: an alias of SMADDL.

[SSBB](#): Speculative Store Bypass Barrier.

[ST2G](#): Store Allocation Tags.

STADD, STADDL: Atomic add on word or doubleword in memory, without return: an alias of LDADD, LDADDA, LDADDAL, LDADDL.

STADDB, STADDLB: Atomic add on byte in memory, without return: an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB.

STADDH, STADDLH: Atomic add on halfword in memory, without return: an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH.

STCLR, STCLRL: Atomic bit clear on word or doubleword in memory, without return: an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL.

STCLRB, STCLRLB: Atomic bit clear on byte in memory, without return: an alias of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.

STCLRH, STCLRLH: Atomic bit clear on halfword in memory, without return: an alias of LDCLRH, LDCLRAB, LDCLRALB, LDCLRLH.

STEOR, STEORL: Atomic exclusive OR on word or doubleword in memory, without return: an alias of LDEOR, LDEORA, LDEORAL, LDEORL.

STEORB, STEORLB: Atomic exclusive OR on byte in memory, without return: an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB.

STEORH, STEORLH: Atomic exclusive OR on halfword in memory, without return: an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH.

**STG: Store Allocation Tag.**

**STGP: Store Allocation Tag and Pair of registers.**

**STGV: Store Tag Vector.**

**STLLR: Store LORelease Register.**

**STLLRB: Store LORelease Register Byte.**

**STLLRH: Store LORelease Register Halfword.**

**STLR: Store-Release Register.**

**STLRB: Store-Release Register Byte.**

**STLRH: Store-Release Register Halfword.**

**STLUR: Store-Release Register (unscaled).**

**STLURB: Store-Release Register Byte (unscaled).**

**STLURH: Store-Release Register Halfword (unscaled).**

**STLXP: Store-Release Exclusive Pair of registers.**

**STLXR: Store-Release Exclusive Register.**

**STLXRB: Store-Release Exclusive Register Byte.**

**STLXRH: Store-Release Exclusive Register Halfword.**

**STNP: Store Pair of Registers, with non-temporal hint.**

**STP: Store Pair of Registers.**

**STR (immediate): Store Register (immediate).**

**STR (register): Store Register (register).**

**STRB (immediate): Store Register Byte (immediate).**

**STRB (register): Store Register Byte (register).**

**STRH (immediate): Store Register Halfword (immediate).**

**STRH (register): Store Register Halfword (register).**

STSET, STSETL: Atomic bit set on word or doubleword in memory, without return: an alias of LDSET, LDSETA, LDSETAL, LDSETL.

STSETB, STSETLB: Atomic bit set on byte in memory, without return: an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB.

STSETH, STSETLH: Atomic bit set on halfword in memory, without return: an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH.

STSMAX, STSMAXL: Atomic signed maximum on word or doubleword in memory, without return: an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.

STSMAXB, STSMAXLB: Atomic signed maximum on byte in memory, without return: an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.

STSMAXH, STSMAXLH: Atomic signed maximum on halfword in memory, without return: an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.

STSMIN, STSMINL: Atomic signed minimum on word or doubleword in memory, without return: an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.

STSMINB, STSMINLB: Atomic signed minimum on byte in memory, without return: an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.

STSMINH, STSMINLH: Atomic signed minimum on halfword in memory, without return: an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.

[STTR](#): Store Register (unprivileged).

[STTRB](#): Store Register Byte (unprivileged).

[STTRH](#): Store Register Halfword (unprivileged).

STUMAX, STUMAXL: Atomic unsigned maximum on word or doubleword in memory, without return: an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.

STUMAXB, STUMAXLB: Atomic unsigned maximum on byte in memory, without return: an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.

STUMAXH, STUMAXLH: Atomic unsigned maximum on halfword in memory, without return: an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.

STUMIN, STUMINL: Atomic unsigned minimum on word or doubleword in memory, without return: an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.

STUMINB, STUMINLB: Atomic unsigned minimum on byte in memory, without return: an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.

STUMINH, STUMINLH: Atomic unsigned minimum on halfword in memory, without return: an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.

[STUR](#): Store Register (unscaled).

[STURB](#): Store Register Byte (unscaled).

[STURH](#): Store Register Halfword (unscaled).

[STXP](#): Store Exclusive Pair of registers.

[STXR](#): Store Exclusive Register.

[STXRB](#): Store Exclusive Register Byte.

[STXRH](#): Store Exclusive Register Halfword.

[STZ2G](#): Store Allocation Tags, Zeroing.

[STZG](#): Store Allocation Tag, Zeroing.

[SUB \(extended register\)](#): Subtract (extended register).

[SUB \(immediate\)](#): Subtract (immediate).

[SUB \(shifted register\)](#): Subtract (shifted register).

[SUBG](#): Subtract with Tag.

[SUBP](#): Subtract Pointer.

[SUBPS](#): Subtract Pointer, setting Flags.

[SUBS \(extended register\)](#): Subtract (extended register), setting flags.

[SUBS \(immediate\)](#): Subtract (immediate), setting flags.

[SUBS \(shifted register\)](#): Subtract (shifted register), setting flags.

SVC: Supervisor Call.

[SWP, SWPA, SWPAL, SWPL](#): Swap word or doubleword in memory.

[SWPB, SWPAB, SWPALB, SWPLB](#): Swap byte in memory.

[SWPH, SWPAH, SWPALH, SWPLH](#): Swap halfword in memory.

SXTB: Signed Extend Byte: an alias of SBFM.

SXTH: Sign Extend Halfword: an alias of SBFM.

SXTW: Sign Extend Word: an alias of SBFM.

[SYS](#): System instruction.

SYSL: System instruction with result.

[TBNZ](#): Test bit and Branch if Nonzero.

[TBZ](#): Test bit and Branch if Zero.

TLBI: TLB Invalidate operation: an alias of SYS.

[TSB CSYNC](#): Trace Synchronization Barrier.

TST (immediate): Test bits (immediate): an alias of ANDS (immediate).

TST (shifted register): Test (shifted register): an alias of ANDS (shifted register).

UBFIZ: Unsigned Bitfield Insert in Zero: an alias of UBFM.

[UBFM](#): Unsigned Bitfield Move.

[UBFX](#): Unsigned Bitfield Extract: an alias of UBFM.

[UDE](#): Permanently Undefined.

UDIV: Unsigned Divide.

UMADDL: Unsigned Multiply-Add Long.

[UMNEGL](#): Unsigned Multiply-Negate Long: an alias of UMSUBL.

UMSUBL: Unsigned Multiply-Subtract Long.

UMULH: Unsigned Multiply High.

UMULL: Unsigned Multiply Long: an alias of UMADDL.

UXTB: Unsigned Extend Byte: an alias of UBFM.

UXTH: Unsigned Extend Halfword: an alias of UBFM.

[WFE](#): Wait For Event.

[WFI](#): Wait For Interrupt.

[XAFlag](#): Convert floating-point condition flags from external format to ARM format.

[XPACD, XPACI, XPACLRI](#): Strip Pointer Authentication Code.

[YIELD](#): YIELD.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:20:09](#)~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

(old)

htmldiff from-

(new)

## A64 -- SIMD and Floating-point Instructions (alphabetic order)

[ABS](#): Absolute value (vector).

[ADD \(vector\)](#): Add (vector).

[ADDHN, ADDHN2](#): Add returning High Narrow.

[ADDP \(scalar\)](#): Add Pair of elements (scalar).

[ADDP \(vector\)](#): Add Pairwise (vector).

[ADDV](#): Add across Vector.

[AESD](#): AES single round decryption.

[AESE](#): AES single round encryption.

[AESIMC](#): AES inverse mix columns.

[AESMC](#): AES mix columns.

AND (vector): Bitwise AND (vector).

[BCAX](#): Bit Clear and XOR.

[BIC \(vector, immediate\)](#): Bitwise bit Clear (vector, immediate).

BIC (vector, register): Bitwise bit Clear (vector, register).

BIF: Bitwise Insert if False.

BIT: Bitwise Insert if True.

BSL: Bitwise Select.

[CLS \(vector\)](#): Count Leading Sign bits (vector).

[CLZ \(vector\)](#): Count Leading Zero bits (vector).

[CMEQ \(register\)](#): Compare bitwise Equal (vector).

[CMEQ \(zero\)](#): Compare bitwise Equal to zero (vector).

[CMGE \(register\)](#): Compare signed Greater than or Equal (vector).

[CMGE \(zero\)](#): Compare signed Greater than or Equal to zero (vector).

[CMGT \(register\)](#): Compare signed Greater than (vector).

[CMGT \(zero\)](#): Compare signed Greater than zero (vector).

[CMHI \(register\)](#): Compare unsigned Higher (vector).

[CMHS \(register\)](#): Compare unsigned Higher or Same (vector).

[CMLE \(zero\)](#): Compare signed Less than or Equal to zero (vector).

[CMLT \(zero\)](#): Compare signed Less than zero (vector).

[CMTST](#): Compare bitwise Test bits nonzero (vector).

[CNT](#): Population Count per byte.

[DUP \(element\)](#): Duplicate vector element to vector or scalar.

[DUP \(general\)](#): Duplicate general-purpose register to vector.



EOR (vector): Bitwise Exclusive OR (vector).

[EOR3](#): Three-way Exclusive OR.

[EXT](#): Extract vector from pair of vectors.

[FABD](#): Floating-point Absolute Difference (vector).

[FABS \(scalar\)](#): Floating-point Absolute value (scalar).

[FABS \(vector\)](#): Floating-point Absolute value (vector).

[FACGE](#): Floating-point Absolute Compare Greater than or Equal (vector).

[FACGT](#): Floating-point Absolute Compare Greater than (vector).

[FADD \(scalar\)](#): Floating-point Add (scalar).

[FADD \(vector\)](#): Floating-point Add (vector).

[FADDP \(scalar\)](#): Floating-point Add Pair of elements (scalar).

[FADDP \(vector\)](#): Floating-point Add Pairwise (vector).

[FCADD](#): Floating-point Complex Add.

[FCCMP](#): Floating-point Conditional quiet Compare (scalar).

[FCCMPE](#): Floating-point Conditional signaling Compare (scalar).

[FCMEQ \(register\)](#): Floating-point Compare Equal (vector).

[FCMEQ \(zero\)](#): Floating-point Compare Equal to zero (vector).

[FCMGE \(register\)](#): Floating-point Compare Greater than or Equal (vector).

[FCMGE \(zero\)](#): Floating-point Compare Greater than or Equal to zero (vector).

[FCMGT \(register\)](#): Floating-point Compare Greater than (vector).

[FCMGT \(zero\)](#): Floating-point Compare Greater than zero (vector).

[FCMLA](#): Floating-point Complex Multiply Accumulate.

[FCMLA \(by element\)](#): Floating-point Complex Multiply Accumulate (by element).

[FCMLE \(zero\)](#): Floating-point Compare Less than or Equal to zero (vector).

[FCMLT \(zero\)](#): Floating-point Compare Less than zero (vector).

[FCMP](#): Floating-point quiet Compare (scalar).

[FCMPE](#): Floating-point signaling Compare (scalar).

[FCSEL](#): Floating-point Conditional Select (scalar).

[FCVT](#): Floating-point Convert precision (scalar).

[FCVTAS \(scalar\)](#): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).

[FCVTAS \(vector\)](#): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).

[FCVTAU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar).

[FCVTAU \(vector\)](#): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).

FCVTL, FCVTL2: Floating-point Convert to higher precision Long (vector).

[FCVTMS \(scalar\)](#): Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).

[FCVTMS \(vector\)](#): Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).

[FCVTMU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).

[FCVTMU \(vector\)](#): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).

FCVTN, FCVTN2: Floating-point Convert to lower precision Narrow (vector).

[FCVTNS \(scalar\)](#): Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar).

[FCVTNS \(vector\)](#): Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector).

[FCVTNU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar).

[FCVTNU \(vector\)](#): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).

[FCVTPS \(scalar\)](#): Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).

[FCVTPS \(vector\)](#): Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).

[FCVTPU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).

[FCVTPU \(vector\)](#): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).

[FCVTXN, FCVTXN2](#): Floating-point Convert to lower precision Narrow, rounding to odd (vector).

[FCVTZS \(scalar, fixed-point\)](#): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).

[FCVTZS \(scalar, integer\)](#): Floating-point Convert to Signed integer, rounding toward Zero (scalar).

[FCVTZS \(vector, fixed-point\)](#): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).

[FCVTZS \(vector, integer\)](#): Floating-point Convert to Signed integer, rounding toward Zero (vector).

[FCVTZU \(scalar, fixed-point\)](#): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).

[FCVTZU \(scalar, integer\)](#): Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).

[FCVTZU \(vector, fixed-point\)](#): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).

[FCVTZU \(vector, integer\)](#): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).

[FDIV \(scalar\)](#): Floating-point Divide (scalar).

[FDIV \(vector\)](#): Floating-point Divide (vector).

[FJCVTZS](#): Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.

[FMADD](#): Floating-point fused Multiply-Add (scalar).

[FMAX \(scalar\)](#): Floating-point Maximum (scalar).

[FMAX \(vector\)](#): Floating-point Maximum (vector).

[FMAXNM \(scalar\)](#): Floating-point Maximum Number (scalar).

[FMAXNM \(vector\)](#): Floating-point Maximum Number (vector).

[FMAXNMP \(scalar\)](#): Floating-point Maximum Number of Pair of elements (scalar).

[FMAXNMP \(vector\)](#): Floating-point Maximum Number Pairwise (vector).

[FMAXNMV](#): Floating-point Maximum Number across Vector.

[FMAXP \(scalar\)](#): Floating-point Maximum of Pair of elements (scalar).

[FMAXP \(vector\)](#): Floating-point Maximum Pairwise (vector).

[FMAXV](#): Floating-point Maximum across Vector.

[FMIN \(scalar\)](#): Floating-point Minimum (scalar).

[FMIN \(vector\)](#): Floating-point minimum (vector).

[FMINNM \(scalar\)](#): Floating-point Minimum Number (scalar).

[FMINNM \(vector\)](#): Floating-point Minimum Number (vector).

[FMINNMP \(scalar\)](#): Floating-point Minimum Number of Pair of elements (scalar).

[FMINNMP \(vector\)](#): Floating-point Minimum Number Pairwise (vector).

[FMINNMV](#): Floating-point Minimum Number across Vector.

[FMINP \(scalar\)](#): Floating-point Minimum of Pair of elements (scalar).

[FMINP \(vector\)](#): Floating-point Minimum Pairwise (vector).

[FMINV](#): Floating-point Minimum across Vector.

[FMLA \(by element\)](#): Floating-point fused Multiply-Add to accumulator (by element).

[FMLA \(vector\)](#): Floating-point fused Multiply-Add to accumulator (vector).

[FMLAL, FMLAL2 \(by element\)](#): Floating-point fused Multiply-Add Long to accumulator (by element).

[FMLAL, FMLAL2 \(vector\)](#): Floating-point fused Multiply-Add Long to accumulator (vector).

[FMLS \(by element\)](#): Floating-point fused Multiply-Subtract from accumulator (by element).

[FMLS \(vector\)](#): Floating-point fused Multiply-Subtract from accumulator (vector).

[FMLS, FMLS2 \(by element\)](#): Floating-point fused Multiply-Subtract Long from accumulator (by element).

[FMLS, FMLS2 \(vector\)](#): Floating-point fused Multiply-Subtract Long from accumulator (vector).

[FMOV \(general\)](#): Floating-point Move to or from general-purpose register without conversion.

[FMOV \(register\)](#): Floating-point Move register without conversion.

[FMOV \(scalar, immediate\)](#): Floating-point move immediate (scalar).

[FMOV \(vector, immediate\)](#): Floating-point move immediate (vector).

[FMSUB](#): Floating-point Fused Multiply-Subtract (scalar).

[FMUL \(by element\)](#): Floating-point Multiply (by element).

[FMUL \(scalar\)](#): Floating-point Multiply (scalar).

[FMUL \(vector\)](#): Floating-point Multiply (vector).

[FMULX](#): Floating-point Multiply extended.

[FMULX \(by element\)](#): Floating-point Multiply extended (by element).

[FNEG \(scalar\)](#): Floating-point Negate (scalar).

[FNEG \(vector\)](#): Floating-point Negate (vector).

[FNMADD](#): Floating-point Negated fused Multiply-Add (scalar).

[FNMSUB](#): Floating-point Negated fused Multiply-Subtract (scalar).

[FNMUL \(scalar\)](#): Floating-point Multiply-Negate (scalar).

[FRECPE](#): Floating-point Reciprocal Estimate.

[FRECPS](#): Floating-point Reciprocal Step.

[FRECPX](#): Floating-point Reciprocal exponent (scalar).

[FRINT32X \(scalar\)](#): Floating-point Round to 32-bit Integer, using current rounding mode (scalar).

[FRINT32X \(vector\)](#): Floating-point Round to 32-bit Integer, using current rounding mode (vector).

[FRINT32Z \(scalar\)](#): Floating-point Round to 32-bit Integer toward Zero (scalar).

[FRINT32Z \(vector\)](#): Floating-point Round to 32-bit Integer toward Zero (vector).

[FRINT64X \(scalar\)](#): Floating-point Round to 64-bit Integer, using current rounding mode (scalar).

[FRINT64X \(vector\)](#): Floating-point Round to 64-bit Integer, using current rounding mode (vector).

[FRINT64Z \(scalar\)](#): Floating-point Round to 64-bit Integer toward Zero (scalar).

[FRINT64Z \(vector\)](#): Floating-point Round to 64-bit Integer toward Zero (vector).

[FRINTA \(scalar\)](#): Floating-point Round to Integral, to nearest with ties to Away (scalar).

[FRINTA \(vector\)](#): Floating-point Round to Integral, to nearest with ties to Away (vector).

[FRINTI \(scalar\)](#): Floating-point Round to Integral, using current rounding mode (scalar).

[FRINTI \(vector\)](#): Floating-point Round to Integral, using current rounding mode (vector).

[FRINTM \(scalar\)](#): Floating-point Round to Integral, toward Minus infinity (scalar).

[FRINTM \(vector\)](#): Floating-point Round to Integral, toward Minus infinity (vector).

[FRINTN \(scalar\)](#): Floating-point Round to Integral, to nearest with ties to even (scalar).

[FRINTN \(vector\)](#): Floating-point Round to Integral, to nearest with ties to even (vector).

[FRINTP \(scalar\)](#): Floating-point Round to Integral, toward Plus infinity (scalar).

[FRINTP \(vector\)](#): Floating-point Round to Integral, toward Plus infinity (vector).

[FRINTX \(scalar\)](#): Floating-point Round to Integral exact, using current rounding mode (scalar).

[FRINTX \(vector\)](#): Floating-point Round to Integral exact, using current rounding mode (vector).

[FRINTZ \(scalar\)](#): Floating-point Round to Integral, toward Zero (scalar).

[FRINTZ \(vector\)](#): Floating-point Round to Integral, toward Zero (vector).

[FRSQRT](#): Floating-point Reciprocal Square Root Estimate.

[FRSQRTS](#): Floating-point Reciprocal Square Root Step.

[FSQRT \(scalar\)](#): Floating-point Square Root (scalar).

[FSQRT \(vector\)](#): Floating-point Square Root (vector).

[FSUB \(scalar\)](#): Floating-point Subtract (scalar).

[FSUB \(vector\)](#): Floating-point Subtract (vector).

[INS \(element\)](#): Insert vector element from another vector element.

[INS \(general\)](#): Insert vector element from general-purpose register.

[LD1 \(multiple structures\)](#): Load multiple single-element structures to one, two, three, or four registers.

[LD1 \(single structure\)](#): Load one single-element structure to one lane of one register.

[LD1R](#): Load one single-element structure and Replicate to all lanes (of one register).

[LD2 \(multiple structures\)](#): Load multiple 2-element structures to two registers.

[LD2 \(single structure\)](#): Load single 2-element structure to one lane of two registers.

[LD2R](#): Load single 2-element structure and Replicate to all lanes of two registers.

[LD3 \(multiple structures\)](#): Load multiple 3-element structures to three registers.

[LD3 \(single structure\)](#): Load single 3-element structure to one lane of three registers).

[LD3R](#): Load single 3-element structure and Replicate to all lanes of three registers.

[LD4 \(multiple structures\)](#): Load multiple 4-element structures to four registers.

[LD4 \(single structure\)](#): Load single 4-element structure to one lane of four registers.

[LD4R](#): Load single 4-element structure and Replicate to all lanes of four registers.

[LDNP \(SIMD&FP\)](#): Load Pair of SIMD&FP registers, with Non-temporal hint.

[LDP \(SIMD&FP\)](#): Load Pair of SIMD&FP registers.

[LDR \(immediate, SIMD&FP\)](#): Load SIMD&FP Register (immediate offset).

[LDR \(literal, SIMD&FP\)](#): Load SIMD&FP Register (PC-relative literal).

[LDR \(register, SIMD&FP\)](#): Load SIMD&FP Register (register offset).

[LDUR \(SIMD&FP\)](#): Load SIMD&FP Register (unscaled offset).

[MLA \(by element\)](#): Multiply-Add to accumulator (vector, by element).

[MLA \(vector\)](#): Multiply-Add to accumulator (vector).

[MLS \(by element\)](#): Multiply-Subtract from accumulator (vector, by element).

[MLS \(vector\)](#): Multiply-Subtract from accumulator (vector).

MOV (element): Move vector element to another vector element: an alias of INS (element).

MOV (from general): Move general-purpose register to a vector element: an alias of INS (general).

MOV (scalar): Move vector element to scalar: an alias of DUP (element).

MOV (to general): Move vector element to general-purpose register: an alias of UMOV.

MOV (vector): Move vector: an alias of ORR (vector, register).

[MOVI](#): Move Immediate (vector).

[MUL \(by element\)](#): Multiply (vector, by element).

[MUL \(vector\)](#): Multiply (vector).

MVN: Bitwise NOT (vector): an alias of NOT.

[MVNI](#): Move inverted Immediate (vector).

[NEG \(vector\)](#): Negate (vector).

NOT: Bitwise NOT (vector).

ORN (vector): Bitwise inclusive OR NOT (vector).

[ORR \(vector, immediate\)](#): Bitwise inclusive OR (vector, immediate).

ORR (vector, register): Bitwise inclusive OR (vector, register).

[PMUL](#): Polynomial Multiply.

[PMULL, PMULL2](#): Polynomial Multiply Long.

[RADDHN, RADDHN2](#): Rounding Add returning High Narrow.

[RAXI](#): Rotate and Exclusive OR.

RBIT (vector): Reverse Bit order (vector).

[REV16 \(vector\)](#): Reverse elements in 16-bit halfwords (vector).

[REV32 \(vector\)](#): Reverse elements in 32-bit words (vector).

[REV64](#): Reverse elements in 64-bit doublewords (vector).

[RSHRN](#), [RSHRN2](#): Rounding Shift Right Narrow (immediate).

[RSUBHN](#), [RSUBHN2](#): Rounding Subtract returning High Narrow.

[SABA](#): Signed Absolute difference and Accumulate.

[SABAL](#), [SABAL2](#): Signed Absolute difference and Accumulate Long.

[SABD](#): Signed Absolute Difference.

[SABDL](#), [SABDL2](#): Signed Absolute Difference Long.

[SADALP](#): Signed Add and Accumulate Long Pairwise.

[SADDL](#), [SADDL2](#): Signed Add Long (vector).

[SADDLP](#): Signed Add Long Pairwise.

[SADDLV](#): Signed Add Long across Vector.

[SADDW](#), [SADDW2](#): Signed Add Wide.

[SCVTF \(scalar, fixed-point\)](#): Signed fixed-point Convert to Floating-point (scalar).

[SCVTF \(scalar, integer\)](#): Signed integer Convert to Floating-point (scalar).

[SCVTF \(vector, fixed-point\)](#): Signed fixed-point Convert to Floating-point (vector).

[SCVTF \(vector, integer\)](#): Signed integer Convert to Floating-point (vector).

[SDOT \(by element\)](#): Dot Product signed arithmetic (vector, by element).

[SDOT \(vector\)](#): Dot Product signed arithmetic (vector).

[SHA1C](#): SHA1 hash update (choose).

[SHA1H](#): SHA1 fixed rotate.

[SHA1M](#): SHA1 hash update (majority).

[SHA1P](#): SHA1 hash update (parity).

[SHA1SU0](#): SHA1 schedule update 0.

[SHA1SU1](#): SHA1 schedule update 1.

[SHA256H](#): SHA256 hash update (part 1).

[SHA256H2](#): SHA256 hash update (part 2).

[SHA256SU0](#): SHA256 schedule update 0.

[SHA256SU1](#): SHA256 schedule update 1.

[SHA512H](#): SHA512 Hash update part 1.

[SHA512H2](#): SHA512 Hash update part 2.

[SHA512SU0](#): SHA512 Schedule Update 0.

[SHA512SU1](#): SHA512 Schedule Update 1.

[SHADD](#): Signed Halving Add.

[SHL](#): Shift Left (immediate).

[SHLL](#), [SHLL2](#): Shift Left Long (by element size).

[SHRN](#), [SHRN2](#): Shift Right Narrow (immediate).

[SHSUB](#): Signed Halving Subtract.

[SLI](#): Shift Left and Insert (immediate).

[SM3PARTW1](#): SM3PARTW1.

[SM3PARTW2](#): SM3PARTW2.

[SM3SS1](#): SM3SS1.

[SM3TT1A](#): SM3TT1A.

[SM3TT1B](#): SM3TT1B.

[SM3TT2A](#): SM3TT2A.

[SM3TT2B](#): SM3TT2B.

[SM4E](#): SM4 Encode.

[SM4EKEY](#): SM4 Key.

[SMAX](#): Signed Maximum (vector).

[SMAXP](#): Signed Maximum Pairwise.

[SMAXV](#): Signed Maximum across Vector.

[SMIN](#): Signed Minimum (vector).

[SMINP](#): Signed Minimum Pairwise.

[SMINV](#): Signed Minimum across Vector.

[SMLAL](#), [SMLAL2 \(by element\)](#): Signed Multiply-Add Long (vector, by element).

[SMLAL](#), [SMLAL2 \(vector\)](#): Signed Multiply-Add Long (vector).

[SMLSL](#), [SMLSL2 \(by element\)](#): Signed Multiply-Subtract Long (vector, by element).

[SMLSL](#), [SMLSL2 \(vector\)](#): Signed Multiply-Subtract Long (vector).

[SMOV](#): Signed Move vector element to general-purpose register.

[SMULL](#), [SMULL2 \(by element\)](#): Signed Multiply Long (vector, by element).

[SMULL](#), [SMULL2 \(vector\)](#): Signed Multiply Long (vector).

[SQABS](#): Signed saturating Absolute value.

[SQADD](#): Signed saturating Add.

[SQDMLAL](#), [SQDMLAL2 \(by element\)](#): Signed saturating Doubling Multiply-Add Long (by element).

[SQDMLAL](#), [SQDMLAL2 \(vector\)](#): Signed saturating Doubling Multiply-Add Long.

[SQDMLSL](#), [SQDMLSL2 \(by element\)](#): Signed saturating Doubling Multiply-Subtract Long (by element).

[SQDMLSL](#), [SQDMLSL2 \(vector\)](#): Signed saturating Doubling Multiply-Subtract Long.

[SQDMULH \(by element\)](#): Signed saturating Doubling Multiply returning High half (by element).

[SQDMULH \(vector\)](#): Signed saturating Doubling Multiply returning High half.

[SQDMULL](#), [SQDMULL2 \(by element\)](#): Signed saturating Doubling Multiply Long (by element).

[SQDMULL](#), [SQDMULL2 \(vector\)](#): Signed saturating Doubling Multiply Long.

[SQNEG](#): Signed saturating Negate.

[SQRDMLAH \(by element\)](#): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

[SQRDMLAH \(vector\)](#): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

[SQRDMLSH \(by element\)](#): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).

[SQRDMLSH \(vector\)](#): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).

[SQRDMULH \(by element\)](#): Signed saturating Rounding Doubling Multiply returning High half (by element).

[SQRDMULH \(vector\)](#): Signed saturating Rounding Doubling Multiply returning High half.

[SQRSHL](#): Signed saturating Rounding Shift Left (register).

[SQRSHRN, SQRSHRN2](#): Signed saturating Rounded Shift Right Narrow (immediate).

[SQRSHRUN, SQRSHRUN2](#): Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

[SQSHL \(immediate\)](#): Signed saturating Shift Left (immediate).

[SQSHL \(register\)](#): Signed saturating Shift Left (register).

[SQSHLU](#): Signed saturating Shift Left Unsigned (immediate).

[SQSHRN, SQSHRN2](#): Signed saturating Shift Right Narrow (immediate).

[SQSHRUN, SQSHRUN2](#): Signed saturating Shift Right Unsigned Narrow (immediate).

[SQSUB](#): Signed saturating Subtract.

[SOXTN, SOXTN2](#): Signed saturating extract Narrow.

[SOXTUN, SOXTUN2](#): Signed saturating extract Unsigned Narrow.

[SRHADD](#): Signed Rounding Halving Add.

[SRI](#): Shift Right and Insert (immediate).

[SRSHL](#): Signed Rounding Shift Left (register).

[SRSHR](#): Signed Rounding Shift Right (immediate).

[SRSRA](#): Signed Rounding Shift Right and Accumulate (immediate).

[SSHLL](#): Signed Shift Left (register).

[SSHLL, SSHLL2](#): Signed Shift Left Long (immediate).

[SSHR](#): Signed Shift Right (immediate).

[SSRA](#): Signed Shift Right and Accumulate (immediate).

[SSUBL, SSUBL2](#): Signed Subtract Long.

[SSUBW, SSUBW2](#): Signed Subtract Wide.

[ST1 \(multiple structures\)](#): Store multiple single-element structures from one, two, three, or four registers.

[ST1 \(single structure\)](#): Store a single-element structure from one lane of one register.

[ST2 \(multiple structures\)](#): Store multiple 2-element structures from two registers.

[ST2 \(single structure\)](#): Store single 2-element structure from one lane of two registers.

[ST3 \(multiple structures\)](#): Store multiple 3-element structures from three registers.

[ST3 \(single structure\)](#): Store single 3-element structure from one lane of three registers.

[ST4 \(multiple structures\)](#): Store multiple 4-element structures from four registers.

[ST4 \(single structure\)](#): Store single 4-element structure from one lane of four registers.

[STNP \(SIMD&FP\)](#): Store Pair of SIMD&FP registers, with Non-temporal hint.



[STP \(SIMD&FP\)](#): Store Pair of SIMD&FP registers.

[STR \(immediate, SIMD&FP\)](#): Store SIMD&FP register (immediate offset).

[STR \(register, SIMD&FP\)](#): Store SIMD&FP register (register offset).

[STUR \(SIMD&FP\)](#): Store SIMD&FP register (unscaled offset).

[SUB \(vector\)](#): Subtract (vector).

[SUBHN, SUBHN2](#): Subtract returning High Narrow.

[SUQADD](#): Signed saturating Accumulate of Unsigned value.

SXTL, SXTL2: Signed extend Long: an alias of SSHLL, SSHLL2.

[TBL](#): Table vector Lookup.

[TBX](#): Table vector lookup extension.

[TRN1](#): Transpose vectors (primary).

[TRN2](#): Transpose vectors (secondary).

[UABA](#): Unsigned Absolute difference and Accumulate.

[UABAL, UABAL2](#): Unsigned Absolute difference and Accumulate Long.

[UABD](#): Unsigned Absolute Difference (vector).

[UABDL, UABDL2](#): Unsigned Absolute Difference Long.

[UADALP](#): Unsigned Add and Accumulate Long Pairwise.

[UADDL, UADDL2](#): Unsigned Add Long (vector).

[UADDLP](#): Unsigned Add Long Pairwise.

[UADDLV](#): Unsigned sum Long across Vector.

[UADDW, UADDW2](#): Unsigned Add Wide.

[UCVTF \(scalar, fixed-point\)](#): Unsigned fixed-point Convert to Floating-point (scalar).

[UCVTF \(scalar, integer\)](#): Unsigned integer Convert to Floating-point (scalar).

[UCVTF \(vector, fixed-point\)](#): Unsigned fixed-point Convert to Floating-point (vector).

[UCVTF \(vector, integer\)](#): Unsigned integer Convert to Floating-point (vector).

[UDOT \(by element\)](#): Dot Product unsigned arithmetic (vector, by element).

[UDOT \(vector\)](#): Dot Product unsigned arithmetic (vector).

[UHADD](#): Unsigned Halving Add.

[UHSUB](#): Unsigned Halving Subtract.

[UMAX](#): Unsigned Maximum (vector).

[UMAXP](#): Unsigned Maximum Pairwise.

[UMAXV](#): Unsigned Maximum across Vector.

[UMIN](#): Unsigned Minimum (vector).

[UMINP](#): Unsigned Minimum Pairwise.

[UMINV](#): Unsigned Minimum across Vector.

[UMLAL, UMLAL2 \(by element\)](#): Unsigned Multiply-Add Long (vector, by element).

[UMLAL, UMLAL2 \(vector\)](#): Unsigned Multiply-Add Long (vector).

[UMLSL, UMLSL2 \(by element\)](#): Unsigned Multiply-Subtract Long (vector, by element).

[UMLSL, UMLSL2 \(vector\)](#): Unsigned Multiply-Subtract Long (vector).

[UMOV](#): Unsigned Move vector element to general-purpose register.

[UMULL, UMULL2 \(by element\)](#): Unsigned Multiply Long (vector, by element).

[UMULL, UMULL2 \(vector\)](#): Unsigned Multiply long (vector).

[UQADD](#): Unsigned saturating Add.

[UQRSHL](#): Unsigned saturating Rounding Shift Left (register).

[UQRSHRN, UQRSHRN2](#): Unsigned saturating Rounded Shift Right Narrow (immediate).

[UQSHL \(immediate\)](#): Unsigned saturating Shift Left (immediate).

[UQSHL \(register\)](#): Unsigned saturating Shift Left (register).

[UQSHRN, UQSHRN2](#): Unsigned saturating Shift Right Narrow (immediate).

[UQSUB](#): Unsigned saturating Subtract.

[UQXTN, UQXTN2](#): Unsigned saturating extract Narrow.

[URECPE](#): Unsigned Reciprocal Estimate.

[URHADD](#): Unsigned Rounding Halving Add.

[URSHL](#): Unsigned Rounding Shift Left (register).

[URSHR](#): Unsigned Rounding Shift Right (immediate).

[URSQRTE](#): Unsigned Reciprocal Square Root Estimate.

[URSRA](#): Unsigned Rounding Shift Right and Accumulate (immediate).

[USHL](#): Unsigned Shift Left (register).

[USHLL, USHLL2](#): Unsigned Shift Left Long (immediate).

[USHR](#): Unsigned Shift Right (immediate).

[USQADD](#): Unsigned saturating Accumulate of Signed value.

[USRA](#): Unsigned Shift Right and Accumulate (immediate).

[USUBL, USUBL2](#): Unsigned Subtract Long.

[USUBW, USUBW2](#): Unsigned Subtract Wide.

UXTL, UXTL2: Unsigned extend Long: an alias of USHLL, USHLL2.

[UZP1](#): Unzip vectors (primary).

[UZP2](#): Unzip vectors (secondary).

[XAR](#): Exclusive OR and Rotate.

[XTN, XTN2](#): Extract Narrow.

[ZIP1](#): Zip vectors (primary).

[ZIP2](#): Zip vectors (secondary).

(old)	htmldiff from-	(new)
-------	----------------	-------

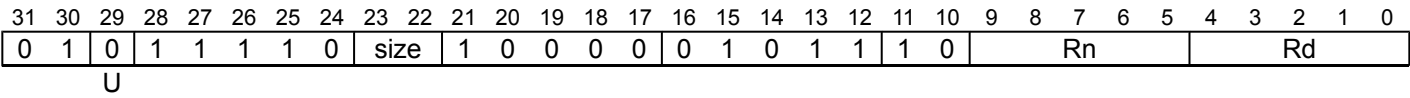
ABS

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



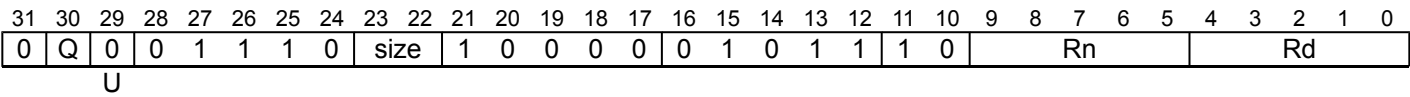
Scalar

ABS `<V><d>`, `<V><n>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector



Vector

ABS `<Vd>.<T>`, `<Vn>.<T>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<V>	Is a width specifier, encoded in “size”:								
<table><tr><th>size</th><th>&lt;V&gt;</th></tr><tr><td>0x</td><td>RESERVED</td></tr><tr><td>10</td><td>RESERVED</td></tr><tr><td>11</td><td>D</td></tr></table>		size	<V>	0x	RESERVED	10	RESERVED	11	D
size	<V>								
0x	RESERVED								
10	RESERVED								
11	D								

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>** Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:04Z2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

(new)

## ADD (extended register)

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	0	1	0	1	1	0	0	1	Rm					option			imm3			Rn				Rd					
op S																															

### 32-bit (sf == 0)

```
ADD <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

### 64-bit (sf == 1)

```
ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED; if shift > 4 then ReservedValue();
```

## Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25-v29.05, AdvSIMD v27.01-v26.0, pseudocode v85-xml-00bet8\_rc3-v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

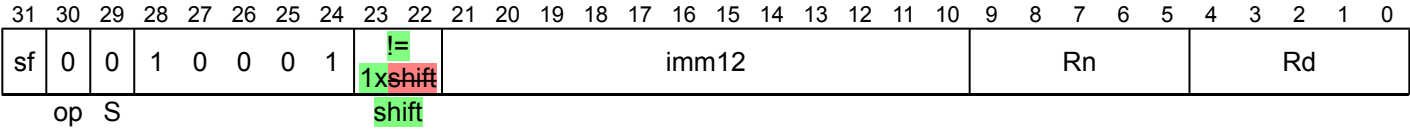
htmldiff from-

(new)

### ADD (immediate)

Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias [MOV \(to/from SP\)](#).



#### 32-bit (sf == 0)

```
ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

#### 64-bit (sf == 1)

```
ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
  when '00' imm = ZeroExtend(imm12, datasize);
  when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
  when '10' SEE "ADDG, SUBG";
  when '11' when '1x' ReservedValue();
```

### Assembler Symbols

- <Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- <shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

shift<0>	<shift>
000	LSL #0
101	LSL #12
1x	RESERVED

### Alias Conditions

Alias	Is preferred when
<a href="#">MOV (to/from SP)</a>	shift == '00' && imm12 == '000000000000' && (Rd == '11111'    Rn == '11111')



## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25-v29.05**, AdvSIMD **v27.01-v26.0**, pseudocode **v85-xml-00bet8\_rc3-v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

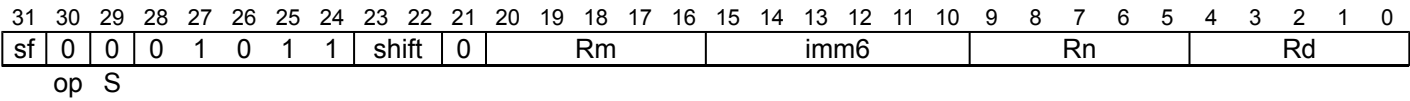
(old)

htmldiff from-

(new)

ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.



32-bit (sf == 0)

```
ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
if shift == '11' then
    ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

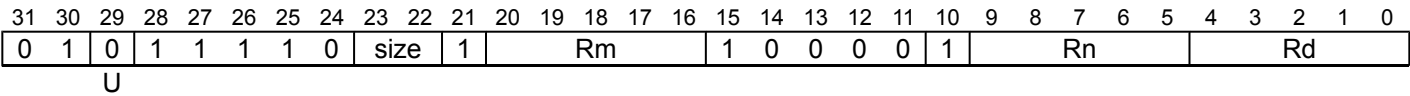
## ADD (vector)

Add (vector). This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

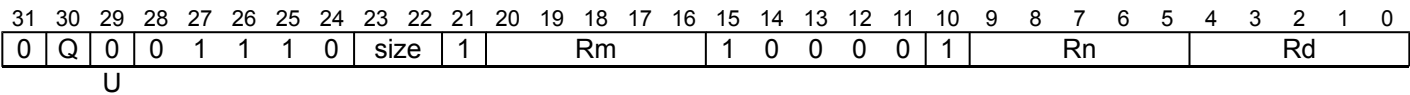


### Scalar

ADD `<V><d>`, `<V><n>`, `<V><m>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

### Vector



### Vector

ADD `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.<T>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

### Assembler Symbols

- <V>

Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D
- <d>

Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d] = result;

```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ADDG

Add with Tag adds an immediate value scaled by the Tag granule to the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR\_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

### Integer (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	1	1	0	uimm6						(0)	(0)	uimm4				Xn				Xd					
																(0)(0)															

### Integer

```
ADDG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>
```

```
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(4) tag_offset = uimm4;
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
boolean ADD = TRUE;
```

### Assembler Symbols

<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
<Xn SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.
<uimm6>	Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
<uimm4>	Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

### Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(4) start_tag = AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AllocationTagAccessIsEnabled() then
    rtag = ChooseNonExcludedTag(start_tag, tag_offset, exclude);
else
    rtag = '0000';

if ADD then
    (result, -) = AddWithCarry(operand1, offset, '0');
else
    (result, -) = AddWithCarry(operand1, NOT(offset), '1');

result = AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

no old file

htmldiff from-

(new)

## ADDHN, ADDHN2

Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are truncated. For rounded results, see [RADDHN](#).

The ADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the ADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						0	1	0	0	0	0	Rn						Rd			
U										o1																					

### Three registers, not all the same type

```
ADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED



<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ADDP (scalar)

Add Pair of elements (scalar). This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	1	0	0	0	1	1	0	1	1	1	0	Rn				Rd						

### Advanced SIMD

ADDP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;

integer esize = 8 << if size != '11' then ReservedValue();

integer esize = 8 << UInt(size);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = ReduceOp_ADD;
```

### Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in “size”:

size	<T>
0x	RESERVED
10	RESERVED
11	2D

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

(old)

htmldiff from-

(new)

## ADDP (vector)

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						1	0	1	1	1	1	Rn						Rd			

### Three registers of the same type

ADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

## ADDS (extended register)

Add (extended register), setting flags, adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias [CMN \(extended register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	0	1	0	1	1	0	0	1	Rm					option			imm3			Rn					Rd				
op S																															

### 32-bit (sf == 0)

```
ADDS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

### 64-bit (sf == 1)

```
ADDS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED; if shift > 4 then ReservedValue();
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn WSP>	Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn SP>	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
<R>	Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m>	Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
<extend>	For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount>

Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Alias Conditions

Alias	Is preferred when
<a href="#">CMN (extended register)</a>	Rd == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcvc;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcvc;

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

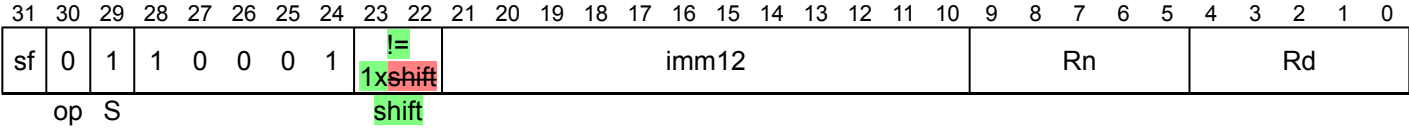
Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

### ADDS (immediate)

Add (immediate), setting flags, adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [CMN \(immediate\)](#).



#### 32-bit (sf == 0)

ADDS <Wd>, <Wn|WSP>, #<imm>{, <shift>}

#### 64-bit (sf == 1)

ADDS <Xd>, <Xn|SP>, #<imm>{, <shift>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
  when '00' imm = ZeroExtend(imm12, datasize);
  when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
  when '10' SEE "ADDG, SUBG";
  when '11' when '1x' ReservedValue();
```

### Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- <shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

shift<0>	<shift>
000	LSL #0
101	LSL #12
1x	RESERVED

### Alias Conditions

Alias	Is preferred when
<a href="#">CMN (immediate)</a>	Rd == '11111'



## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25-v29.05**, AdvSIMD **v27.01-v26.0**, pseudocode **v85-xml-00bet8\_rc3-v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

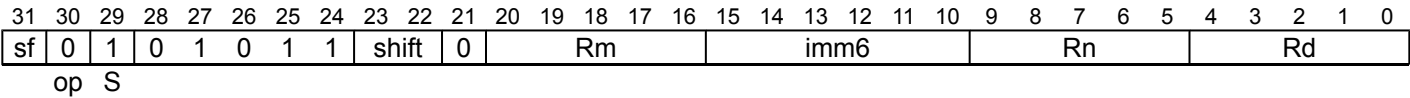
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

### ADDS (shifted register)

Add (shifted register), setting flags, adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [CMN \(shifted register\)](#).



#### 32-bit (sf == 0)

```
ADDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

#### 64-bit (sf == 1)

```
ADDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED; if shift == '11' then

ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

### Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">CMN (shifted register)</a>	Rd == '11111'

## Operation

```

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcw;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcw) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcw;

X[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

(new)

## ADDV

Add across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	1	0	0	0	1	1	0	1	1	1	0		Rn					Rd				

### Advanced SIMD

ADDV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;

integer esize = 8 << if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = ReduceOp_ADD;
```

### Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## AESD

AES single round decryption.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	0	1	1	0	Rn				Rd					

D

## Advanced SIMD

AESD <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAEEExt() then UNDEFINED;
boolean decrypt = (D == '1'); () then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
if decrypt then
    result = AESInvSubBytes(AESInvShiftRows(result));
else
    result = AESSubBytes(AESShiftRows(result));

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## AESE

AES single round encryption.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	0	0	1	0	Rn				Rd					
D																															

## Advanced SIMD

AESE <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAEEExt() then UNDEFINED;
boolean decrypt = (D == '1'); () then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
if decrypt then
    result = AESInvSubBytes(AESInvShiftRows(result));
else
    result = AESSubBytes(AESShiftRows(result));

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESIMC

AES inverse mix columns.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	1	1	1	0	Rn				Rd					
D																															

Advanced SIMD

AESIMC <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAEEExt() then UNDEFINED;
boolean decrypt = (D == '1'); () then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n];
bits(128) result;
if decrypt then
    result = AESInvMixColumns(operand);
else
    result = AESMixColumns(operand);
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.



AESMC

AES mix columns.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	1	0	1	0	Rn			Rd						
D																															

Advanced SIMD

AESMC <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAEEExt() then UNDEFINED;
boolean decrypt = (D == '1'); () then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n];
bits(128) result;
if decrypt then
    result = AESInvMixColumns(operand);
else
    result = AESMixColumns(operand);
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## AND (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	1	0	0	N	immr						imms						Rn						Rd			
opc																															

### 32-bit (sf == 0 && N == 0)

AND <Wd|WSP>, <Wn>, #<imm>

### 64-bit (sf == 1)

AND <Xd|SP>, <Xn>, #<imm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Assembler Symbols

<Wd WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

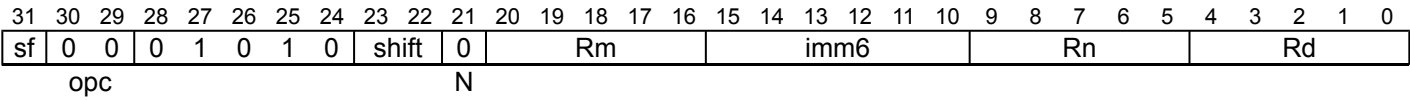
Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

AND (shifted register)

Bitwise AND (shifted register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.



32-bit (sf == 0)

```
AND <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
AND <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

  if sf == '0' && imm6<5> == '1' then UNDEFINED; if sf == '0' && imm6<5> == '1' then
    ReservedValue();

  ShiftType shift_type = DecodeShift(shift);
  integer shift_amount = UInt(imm6);
  boolean invert = (N == '1');
```

Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

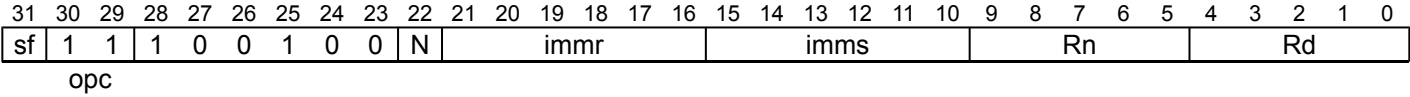
htmldiff from-

(new)

## ANDS (immediate)

Bitwise AND (immediate), setting flags, performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [TST \(immediate\)](#).



### 32-bit (sf == 0 && N == 0)

```
ANDS <Wd>, <Wn>, #<imm>
```

### 64-bit (sf == 1)

```
ANDS <Xd>, <Xn>, #<imm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) =if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <imm>

For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".  
For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

## Alias Conditions

Alias	Is preferred when
<a href="#">TST (immediate)</a>	Rd == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25~~v29.05~~, AdvSIMD v27.01~~v26.0~~, pseudocode v85-xml-00bet8\_rc3~~v35.3~~; Build timestamp: 2018-09-13T13:04:45~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

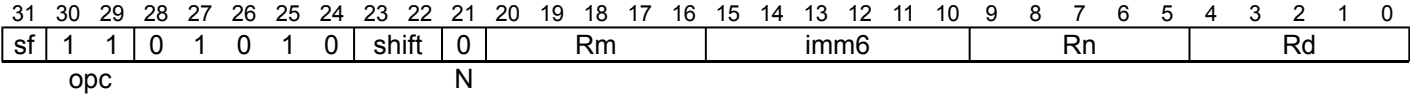
htmldiff from-

(new)

## ANDS (shifted register)

Bitwise AND (shifted register), setting flags, performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [TST \(shifted register\)](#).



### 32-bit (sf == 0)

```
ANDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
ANDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then UNDEFINED; if sf == '0' && imm6<5> == '1' then
ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

## Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,



Alias Conditions

Alias	Is preferred when
<a href="#">TST (shifted register)</a>	Rd == '11111'

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR  result = operand1 OR  operand2;
  when LogicalOp_EOR  result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00~~bet8\_rc3v35.3; Build timestamp: ~~2018-09-13T13:20:18-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## AUTDA, AUTDZA

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A. The address is in the general-purpose register that is specified by <Xd>. The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTDA.
- The value zero, for AUTDZA.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

### Integer (ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	1	1	0	Rn					Rd				

#### AUTDA (Z == 0)

AUTDA <Xd>, <Xn|SP>

#### AUTDZA (Z == 1 && Rn == 11111)

AUTDZA <Xd>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTDA
    if n == 31 then source_is_sp = TRUE;
else // AUTDZA
    if n != 31 then UNDEFINED;() thenUnallocatedEncoding();

if Z == '0' then // AUTDA
    if n == 31 then source_is_sp = TRUE;
else // AUTDZA
    if n != 31 then UnallocatedEncoding();
```

### Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

### Operation

```
if source_is_sp then
    X[d] = AuthDA(X[d], SP[]);
else
    X[d] = AuthDA(X[d], X[n]);
```

## AUTDB, AUTDZB

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B. The address is in the general-purpose register that is specified by <Xd>. The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTDB.
- The value zero, for AUTDZB.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

### Integer (ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	1	1	0	1	0	1	1	0	0	0	0	1	0	0	Z	1	1	1	Rn						Rd					

#### AUTDB (Z == 0)

AUTDB <Xd>, <Xn|SP>

#### AUTDZB (Z == 1 && Rn == 11111)

AUTDZB <Xd>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTDB
    if n == 31 then source_is_sp = TRUE;
else // AUTDZB
    if n != 31 then UNDEFINED;() thenUnallocatedEncoding();

if Z == '0' then // AUTDB
    if n == 31 then source_is_sp = TRUE;
else // AUTDZB
    if n != 31 then UnallocatedEncoding();
```

### Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

### Operation

```
if source_is_sp then
    X[d] = AuthDB(X[d], SP[]);
else
    X[d] = AuthDB(X[d], X[n]);
```

## AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

Authenticate Instruction address, using key A. This instruction authenticates an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by <Xd> for AUTIA and AUTIZA.
- In X17, for AUTIA1716.
- In X30, for AUTIASP and AUTIAZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIA.
- The value zero, for AUTIZA and AUTIAZ.
- In X16, for AUTIA1716.
- In SP, for AUTIASP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

It has encodings from 2 classes: [Integer](#) and [System](#)

### Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	1	0	0	Rn				Rd					

### AUTIA (Z == 0)

AUTIA <Xd>, <Xn|SP>

### AUTIZA (Z == 1 && Rn == 11111)

AUTIZA <Xd>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTIA
    if n == 31 then source_is_sp = TRUE;
else // AUTIZA
    if n != 31 then UNDEFINED; () then UnallocatedEncoding();

if Z == '0' then // AUTIA
    if n == 31 then source_is_sp = TRUE;
else // AUTIZA
    if n != 31 then UnallocatedEncoding();
```

### System

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	1	0	x	1	1	1	1	1
												CRm				op2															

**AUTIA1716 (CRm == 0001 && op2 == 100)**

AUTIA1716

**AUTIASP (CRm == 0011 && op2 == 101)**

AUTIASP

**AUTIAZ (CRm == 0011 && op2 == 100)**

AUTIAZ

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 100' // AUTIAZ
    d = 30;
    n = 31;
  when '0011 101' // AUTIASP
    d = 30;
    source_is_sp = TRUE;
  when '0001 100' // AUTIA1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 010' SEE "PACIB";
  when '0001 110' SEE "AUTIB";
  when '0011 00x' SEE "PACIA";
  when '0011 01x' SEE "PACIB";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPACLRI";
  otherwise SEE "HINT";
```

## Assembler Symbols

**<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

## Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AuthIA(X[d], SP[]);
    else
        X[d] = AuthIA(X[d], X[n]);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3 ; Build timestamp: 2018-09-13T13:20:45Z2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

Authenticate Instruction address, using key B. This instruction authenticates an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by <Xd> for AUTIB and AUTIZB.
- In X17, for AUTIB1716.
- In X30, for AUTIBSP and AUTIBZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIB.
- The value zero, for AUTIZB and AUTIBZ.
- In X16, for AUTIB1716.
- In SP, for AUTIBSP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

It has encodings from 2 classes: [Integer](#) and [System](#)

### Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	1	0	1	Rn				Rd					

### AUTIB (Z == 0)

AUTIB <Xd>, <Xn|SP>

### AUTIZB (Z == 1 && Rn == 11111)

AUTIZB <Xd>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTIB
    if n == 31 then source_is_sp = TRUE;
else // AUTIZB
    if n != 31 then UNDEFINED; () then UnallocatedEncoding();

if Z == '0' then // AUTIB
    if n == 31 then source_is_sp = TRUE;
else // AUTIZB
    if n != 31 then UnallocatedEncoding();
```

### System

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	1	1	x	1	1	1	1	1
												CRm				op2															

## AUTIB1716 (CRm == 0001 && op2 == 110)

AUTIB1716

## AUTIBSP (CRm == 0011 && op2 == 111)

AUTIBSP

## AUTIBZ (CRm == 0011 && op2 == 110)

AUTIBZ

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 110' // AUTIBZ
    d = 30;
    n = 31;
  when '0011 111' // AUTIBSP
    d = 30;
    source_is_sp = TRUE;
  when '0001 110' // AUTIB1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 010' SEE "PACIB";
  when '0001 100' SEE "AUTIA";
  when '0011 00x' SEE "PACIA";
  when '0011 01x' SEE "PACIB";
  when '0011 10x' SEE "AUTIA";
  when '0000 111' SEE "XPACLRI";
```

## Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

## Operation

```
if HavePACExt() then
  if source_is_sp then
    X[d] = AuthIB(X[d], SP);
  else
    X[d] = AuthIB(X[d], X[n]);
```

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## AXFlag

Convert floating-point condition flags from ARM to external format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from a form representing the result of an ARM floating-point scalar compare instruction to an alternative representation required by some software.

### System (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	(0)	(0)	(0)	(0)	0	1	0	1	1	1	1	1

CRm

### System

AXFlag

```
if !HaveFlagFormatExt() then UNDEFINED;
```

### Operation

```
bit N = '0';
bit Z = PSTATE.Z OR PSTATE.V;
bit C = PSTATE.C AND NOT(PSTATE.V);
bit V = '0';
```

```
PSTATE.N = N;
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = V;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.



B.cond

Branch conditionally to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	imm19																		0	cond				

19-bit signed PC-relative branch offset

```
B.<cond> <label>
```

```
bits(64) offset = SignExtend(imm19:'00', 64);
bits(4) condition = cond;
```

Assembler Symbols

- <cond>

Is one of the standard conditions, encoded in the "cond" field in the standard way.
- <label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

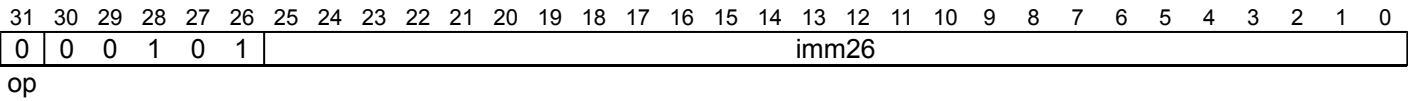
```
if ConditionHolds(condition) then
    BranchTo(PC[] + offset, BranchType DIRBranchType JMP);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

B

Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.



26-bit signed PC-relative branch offset

```
B <label>
```

```
BranchType branch_type = if op == '1' then BranchType_DIRCALLBranchType_CALL else BranchType_DIRBranchT
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler Symbols

<label>

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

```
if branch_type == BranchType_DIRCALLBranchType_CALL then X[30] = PC[] + 4;
BranchTo(PC[] + offset, branch_type);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BCAX

Bit Clear and Exclusive OR performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when [ARMv8.2-SHA](#) is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	1	Rm				0	Ra				Rn				Rd							

### Advanced SIMD

BCAX <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt (Rd);
integer n = UInt (Rn);
integer m = UInt (Rm);
integer a = UInt (Ra);
```

### Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va>	Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Va = V[a];
V[d] = Vn EOR (Vm AND NOT (Va));
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BFM

Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the other bits of the destination register remain unchanged.

This instruction is used by the aliases [BFC](#), [BFI](#), and [BFXIL](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	0	0	1	1	0	N	immr						imms						Rn				Rd					
opc																															

### 32-bit (sf == 0 && N == 0)

BFM <Wd>, <Wn>, #<immr>, #<imms>

### 64-bit (sf == 1 && N == 1)

BFM <Xd>, <Xn>, #<immr>, #<imms>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

case opc of
  when '00' inzero = TRUE;  extend = TRUE;    // SBFM
  when '01' inzero = FALSE; extend = FALSE;   // BFM
  when '10' inzero = TRUE;  extend = FALSE;   // UBFM
  when '11' UNDEFINED;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

R = when '11' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

<imms>

For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Is preferred when
<a href="#">BFC</a>	<code>Rn == '11111' &amp;&amp; <a href="#">UInt</a>(imms) &lt; <a href="#">UInt</a>(immr)</code>
<a href="#">BFI</a>	<code>Rn != '11111' &amp;&amp; <a href="#">UInt</a>(imms) &lt; <a href="#">UInt</a>(immr)</code>
<a href="#">BFXIL</a>	<code><a href="#">UInt</a>(imms) &gt;= <a href="#">UInt</a>(immr)</code>

Operation

```
bits(datasize) dst = if inzero then Zeros() else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## BIC (vector, immediate)

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd				
op												cmode																			

### 16-bit (cmode == 10x1)

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit (cmode == 0xx1)

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11110' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = if Q == '0' then UnallocatedEncoding();
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25**~~v29.05~~, AdvSIMD **v27.01**~~v26.0~~, pseudocode **v85-xml-00bet8\_rc3**~~v35.3~~; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

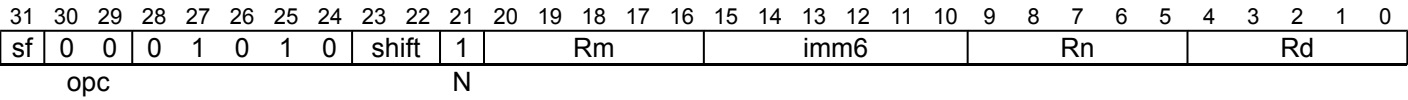
(old)

htmldiff from-

(new)

### BIC (shifted register)

Bitwise Bit Clear (shifted register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.



#### 32-bit (sf == 0)

```
BIC <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

#### 64-bit (sf == 1)

```
BIC <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

  if sf == '0' && imm6<5> == '1' then UNDEFINED; if sf == '0' && imm6<5> == '1' then
    ReservedValue();
  ShiftType shift_type = DecodeShift(shift);
  integer shift_amount = UInt(imm6);
  boolean invert = (N == '1');
```

### Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,



## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR  result = operand1 OR  operand2;
  when LogicalOp_EOR  result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

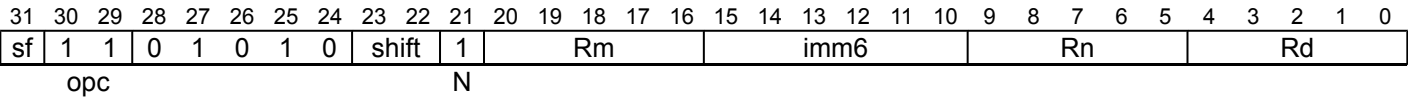
(old)

htmldiff from-

(new)

### BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags, performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.



#### 32-bit (sf == 0)

```
BICS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

#### 64-bit (sf == 1)

```
BICS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then UNDEFINED; if sf == '0' && imm6<5> == '1' then
ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

### Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

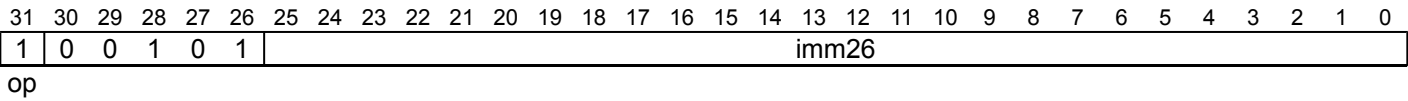
(old)

htmldiff from-

(new)

BL

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.



26-bit signed PC-relative branch offset

```
BL <label>
```

```
BranchType branch_type = if op == '1' then BranchType_DIRCALLBranchType_CALL else BranchType_DIRBranchT
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler Symbols

<label>

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

```
if branch_type == BranchType_DIRCALLBranchType_CALL then X[30] = PC[] + 4;
BranchTo(PC[] + offset, branch_type);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0					Rn		0	0	0	0	0	
Z							op										A M										Rm						

## Integer

BLR <Xn>

```
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UNDEFINED;
elseif pac && !if !pac && m != 0 then UnallocatedEncoding();
elseif pac && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = () then BranchType_INDIRUnallocatedEncoding();
    when '01' branch_type = ();

case op of
    when '00' branch_type = BranchType_INDCALLBranchType_JMP;
    when '10' branch_type = when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n != 31 then UnallocatedEncoding then
            if n != 31 then UNDEFINED;
        ()};

    n = 30;
    source_is_sp = TRUE;
```

## Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
bits(64) target = X[n];
case branch_type of
  whenif pac then
    bits(64) modifier = if source_is_sp then BranchType_INDIR
    if InGuardedPage then
      if n == 16 || n == 17 then
        BTypeNext = '01';
      else
        BTypeNext = '11';
      else
        BTypeNext = '01';
    when BranchType_INDCALL
      BTypeNext = '10';
    when BranchType_RET
      BTypeNext = '00';
if pac then
  bits(64) modifier = if source_is_sp then SP[] else X[m];

  if use_key_a then
    target = AuthIA(target, modifier);
  else
    target = AuthIB(target, modifier);

if branch_type == BranchType_INDCALLBranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

BLRAA, BLRAAZ, BLRAB, BLRABZ

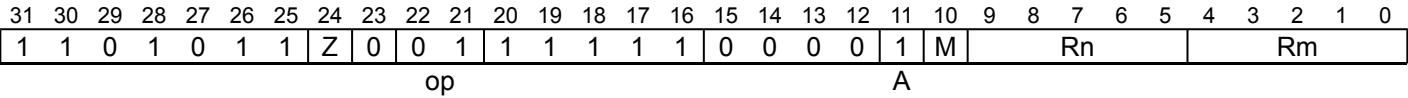
Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4. The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BLRAA and BLRAB.
- The value zero, for BLRAAZ and BLRABZ.

Key A is used for BLRAA and BLRAAZ, and key B is used for BLRAB and BLRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated. The authenticated address is not written back to the general-purpose register.

Integer  
(ARMv8.3)



### Key A, zero modifier (Z == 0 && M == 0 && Rm == 11111)

BLRAAZ <Xn>

### Key A, register modifier (Z == 1 && M == 0)

BLRAA <Xn>, <Xm|SP>

### Key B, zero modifier (Z == 0 && M == 1 && Rm == 11111)

BLRABZ <Xn>

### Key B, register modifier (Z == 1 && M == 1)

BLRAB <Xn>, <Xm|SP>

```
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UNDEFINED;
elseif pac && !if !pac && m != 0 then UnallocatedEncoding();
elseif pac && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = () then BranchType_INDIRUnallocatedEncoding;
    when '01' branch_type = ();

case op of
    when '00' branch_type = BranchType_INDCALLBranchType_JMP;
    when '10' branch_type = when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n != 31 then UnallocatedEncoding then
        if n != 31 then UNDEFINED;
();

n = 30;
source_is_sp = TRUE;
```

## Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.
<Xm SP>	Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.



Operation

```
bits(64) target = X[n];
case branch_type of
  whenif pac then
    bits(64) modifier = if source_is_sp then BranchType_INDIR
    if InGuardedPage then
      if n == 16 || n == 17 then
        BTypeNext = '01';
      else
        BTypeNext = '11';
      else
        BTypeNext = '01';
    when BranchType_INDCALL
      BTypeNext = '10';
    when BranchType_RET
      BTypeNext = '00';
if pac then
  bits(64) modifier = if source_is_sp then SP[] else X[m];

  if use_key_a then
    target = AuthIA(target, modifier);
  else
    target = AuthIB(target, modifier);

if branch_type == BranchType_INDCALLBranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## BR

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0							0	0	0	0	0
Z							op					A					M		Rm													

## Integer

BR <Xn>

```
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UNDEFINED;
elseif pac && !if !pac && m != 0 then UnallocatedEncoding();
elseif pac && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = () then BranchType_INDIR UnallocatedEncoding();
    when '01' branch_type = ();

case op of
    when '00' branch_type = BranchType_INDCALL BranchType_JMP;
    when '10' branch_type = when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n != 31 then UnallocatedEncoding then
            if n != 31 then UNDEFINED;
        then
            n = 30;
            source_is_sp = TRUE;
```

## Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

## Operation

```
bits(64) target = X[n];
case branch_type of
  whenif pac then
    bits(64) modifier = if source_is_sp then BranchType_INDIR
    if InGuardedPage then
      if n == 16 || n == 17 then
        BTypeNext = '01';
      else
        BTypeNext = '11';
      else
        BTypeNext = '01';
    when BranchType_INDCALL
      BTypeNext = '10';
    when BranchType_RET
      BTypeNext = '00';
if pac then
  bits(64) modifier = if source_is_sp then SP[] else X[m];

  if use_key_a then
    target = AuthIA(target, modifier);
  else
    target = AuthIB(target, modifier);

if branch_type == BranchType_INDCALLBranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

# BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:

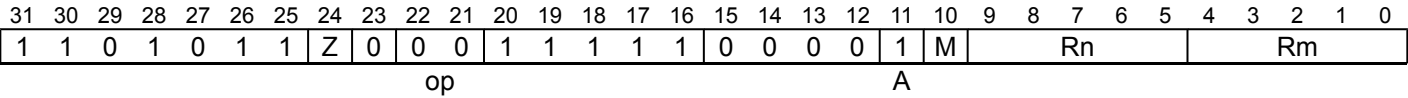
- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

Key A is used for BRAA and BRAAZ, and key B is used for BRAB and BRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

## Integer (ARMv8.3)



### Key A, zero modifier (Z == 0 && M == 0 && Rm == 11111)

BRAAZ <Xn>

### Key A, register modifier (Z == 1 && M == 0)

BRAA <Xn>, <Xm|SP>

### Key B, zero modifier (Z == 0 && M == 1 && Rm == 11111)

BRABZ <Xn>

### Key B, register modifier (Z == 1 && M == 1)

BRAB <Xn>, <Xm|SP>

```
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UNDEFINED;
elseif pac && !if !pac && m != 0 then UnallocatedEncoding();
elseif pac && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = () then BranchType_INDIRUnallocatedEncoding;
    when '01' branch_type = ();

case op of
    when '00' branch_type = BranchType_INDCALLBranchType_JMP;
    when '10' branch_type = when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n != 31 then UnallocatedEncoding then
        if n != 31 then UNDEFINED;
();

n = 30;
source_is_sp = TRUE;
```

## Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.
<Xm SP>	Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

```
bits(64) target = X[n];
case branch_type of
  whenif pac then
    bits(64) modifier = if source_is_sp then BranchType_INDIR
    if InGuardedPage then
      if n == 16 || n == 17 then
        BTypeNext = '01';
      else
        BTypeNext = '11';
      else
        BTypeNext = '01';
    when BranchType_INDCALL
      BTypeNext = '10';
    when BranchType_RET
      BTypeNext = '00';
if pac then
  bits(64) modifier = if source_is_sp then SP[] else X[m];

  if use_key_a then
    target = AuthIA(target, modifier);
  else
    target = AuthIB(target, modifier);

if branch_type == BranchType_INDCALLBranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

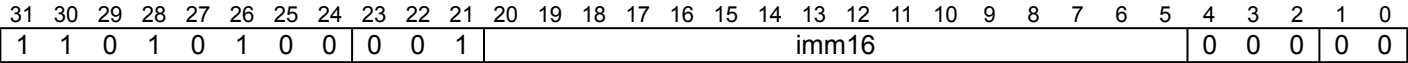
Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

BRK

Breakpoint ~~instruction~~ ~~instruction~~ ~~generates a Breakpoint Instruction exception. The PE records the exception in~~ BRK instruction generates a Breakpoint Instruction exception. The PE records the exception in *ESR\_ELx*, using the EC value 0x3c, and captures the value of the immediate argument in *ESR\_ELx*.ISS.



System

```
BRK #<imm>

bits(16) comment = imm16;
ifbits(16) comment = imm16; HaveBTIEExt() then
    BTypeCompatible = TRUE;
```

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.SoftwareBreakpoint (comment);
```

## BTI

Branch Target Identification. A BTI instruction is used to guard against the execution of instructions which are not the intended target of a branch.

Outside of a guarded memory region, a BTI instruction executes as a NOP. Within a guarded memory region while `PSTATE.BTYPE` != 0b00, a BTI instruction compatible with the current value of `PSTATE.BTYPE` will not generate a Branch Target Exception and will allow execution of subsequent instructions within the memory region.

The operand <targets> passed to a BTI instruction determines the values of `PSTATE.BTYPE` which the BTI instruction is compatible with.

Within a guarded memory region, while `PSTATE.BTYPE`

!= 0b00, all instructions will generate a Branch Target

Exception, other than BRK, BTI, HLT, PACIASP,

and PACIBSP, which may not. See the individual instructions for details.

### System (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	x	x	0	1	1	1	1	1
																CRm				op2											

### System

BTI {<targets>}

```
SystemHintOp op;
```

```
case CRm:op2 of
```

```
  when '0000 000' op = SystemHintOp_NOP;
```

```
  when '0000 001' op = SystemHintOp_YIELD;
```

```
  when '0000 010' op = SystemHintOp_WFE;
```

```
  when '0000 011' op = SystemHintOp_WFI;
```

```
  when '0000 100' op = SystemHintOp_SEV;
```

```
  when '0000 101' op = SystemHintOp_SEVL;
```

```
  when '0000 111'
```

```
    SEE "XPACLR1";
```

```
  when '0001 xxx'
```

```
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
```

```
  when '0010 000'
```

```
    if !HaveRASExt() then EndOfInstruction();
```

```
// Instruction executes as NOP
```

```
    op = SystemHintOp_ESB;
```

```
  when '0010 001'
```

```
    if !HaveStatisticalProfiling() then EndOfInstruction();
```

```
// Instruction executes as NOP
```

```
    op = SystemHintOp_PSB;
```

```
  when '0010 010'
```

```
    if !HaveSelfHostedTrace() then EndOfInstruction();
```

```
// Instruction executes as NOP
```

```
    op = SystemHintOp_TSB;
```

```
  when '0010 100'
```

```
    op = SystemHintOp_CSDB;
```

```
  when '0011 xxx'
```

```
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
```

```
  when '0100 xx0'
```

```
    op = SystemHintOp_BTI;
```

```
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
```

```
  otherwise EndOfInstruction();
```

```
// Instruction executes as NOP
```

### Assembler Symbols

<targets> Is the type of indirection, encoded in “op2<2:1>”:



op2<2:1>	<targets>
00	(omitted)
01	c
10	j
11	jc

## Operation

```

case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, TRUE\);
        WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, FALSE\);
        WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing

```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file

htmldiff from-

(new)

## CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASA and CASAL load from memory with acquire semantics.
- CASL and CASAL store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, or <Xs>, is restored to the value held in the register before the instruction was executed.

### No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	L	1	Rs				o0	1	1	1	1	1	Rn				Rt						

size

### 32-bit CAS (size == 10 && L == 0 && o0 == 0)

```
CAS <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

### 32-bit CASA (size == 10 && L == 1 && o0 == 0)

```
CASA <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

### 32-bit CASAL (size == 10 && L == 1 && o0 == 1)

```
CASAL <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

### 32-bit CASL (size == 10 && L == 0 && o0 == 1)

```
CASL <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

### 64-bit CAS (size == 11 && L == 0 && o0 == 0)

```
CAS <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

### 64-bit CASA (size == 11 && L == 1 && o0 == 0)

```
CASA <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

### 64-bit CASAL (size == 11 && L == 1 && o0 == 1)

```
CASAL <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

### 64-bit CASL (size == 11 && L == 0 && o0 == 1)

```
CASL <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer n = () then UnallocatedEncoding();
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRWAccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRWAccType_ORDEREDRW else AccType_ATOMICRW;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if comparevalue = HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, datasize DIV 8, stacctype] = newvalue;

X[s] = ZeroExtend(data, regsize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

### No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	L	1	Rs				o0	1	1	1	1	1	Rn				Rt						
size																															

### CASAB (L == 1 && o0 == 0)

CASAB <Ws>, <Wt>, [<Xn|SP>{, #0}]

### CASALB (L == 1 && o0 == 1)

CASALB <Ws>, <Wt>, [<Xn|SP>{, #0}]

### CASB (L == 0 && o0 == 0)

CASB <Ws>, <Wt>, [<Xn|SP>{, #0}]

### CASLB (L == 0 && o0 == 1)

CASLB <Ws>, <Wt>, [<Xn|SP>{, #0}]

```

if !HaveAtomicExt() then UNDEFINED;

integer n = () then UnallocatedEncoding();

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType ORDEREDATOMICRW else AccType ATOMICRW;
AccType stacctype = if o0 == '1' then AccType ORDEREDATOMICRW else AccType ATOMICRW;

```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if comparevalue = HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, datasize DIV 8, stacctype] = newvalue;

X[s] = ZeroExtend(data, regsize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

### No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	L	1					Rs	o0	1	1	1	1	1										Rt
size																															

### CASAH (L == 1 && o0 == 0)

CASAH <Ws>, <Wt>, [<Xn|SP>{, #0}]

### CASALH (L == 1 && o0 == 1)

CASALH <Ws>, <Wt>, [<Xn|SP>{, #0}]

### CASH (L == 0 && o0 == 0)

CASH <Ws>, <Wt>, [<Xn|SP>{, #0}]

### CASLH (L == 0 && o0 == 1)

CASLH <Ws>, <Wt>, [<Xn|SP>{, #0}]

```

if !HaveAtomicExt() then UNDEFINED;

integer n = () then UnallocatedEncoding();

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType ORDEREDATOMICRW else AccType ATOMICRW;
AccType stacctype = if o0 == '1' then AccType ORDEREDATOMICRW else AccType ATOMICRW;

```

### Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if comparevalue = HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, datasize DIV 8, stacctype] = newvalue;

X[s] = ZeroExtend(data, regsize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- CASPA and CASPAL load from memory with acquire semantics.
- CASPL and CASPAL store to memory with release semantics.
- CAS has no memory ordering requirements.

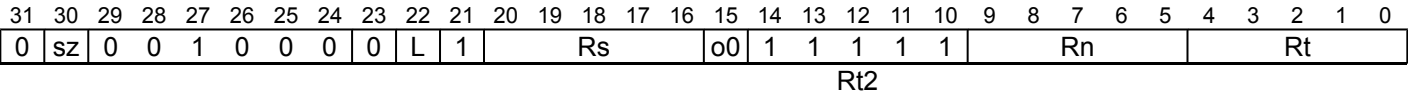
For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is <Ws> and <W(s+1)>, or <Xs> and <X(s+1)>, are restored to the values held in the registers before the instruction was executed.

### No offset (ARMv8.1)



### 32-bit CASP (sz == 0 && L == 0 && o0 == 0)

CASP <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]

### 32-bit CASPA (sz == 0 && L == 1 && o0 == 0)

CASPA <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]

### 32-bit CASPAL (sz == 0 && L == 1 && o0 == 1)

CASPAL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]

### 32-bit CASPL (sz == 0 && L == 0 && o0 == 1)

CASPL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]

### 64-bit CASP (sz == 1 && L == 0 && o0 == 0)

CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]

### 64-bit CASPA (sz == 1 && L == 1 && o0 == 0)

CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]

### 64-bit CASPAL (sz == 1 && L == 1 && o0 == 1)

CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]

### 64-bit CASPL (sz == 1 && L == 0 && o0 == 1)

CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]

```
if !HaveAtomicExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;

integer n = () then UnallocatedEncoding();
if Rs<0> == '1' then UnallocatedEncoding();
if Rt<0> == '1' then UnallocatedEncoding();

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 32 << UInt(sz);
integer regsize = datasize;
AccType ldacctype = if L == '1' then AccType ORDEREDATOMICRWAccType_ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if o0 == '1' then AccType ORDEREDATOMICRWAccType_ORDEREDRW else AccType ATOMICRW;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Ws> must be an even-numbered register.
<W(s+1)>	Is the 32-bit name of the second general-purpose register to be compared and loaded.
<Wt>	Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Wt> must be an even-numbered register.
<W(t+1)>	Is the 32-bit name of the second general-purpose register to be conditionally stored.
<Xs>	Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.

<X(s+1)>	Is the 64-bit name of the second general-purpose register to be compared and loaded.
<Xt>	Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
<X(t+1)>	Is the 64-bit name of the second general-purpose register to be conditionally stored.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

bits(datasize) s1 = X[s];
bits(datasize) s2 = X[s+1];
bits(datasize) t1 = X[t];
bits(datasize) t2 = X[t+1];
comparevalue = if BigEndian() then s1:s2 else s2:s1;
newvalue      = if BigEndian() then t1:t2 else t2:t1;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, (2 * datasize) DIV 8, ldacctype];
if data == comparevalue then
    Mem[address, (2 * datasize) DIV 8, stacctype] = newvalue;

if BigEndian() then
    X[s] = ZeroExtend(data<2*datasize-1:datasize>, regsize);
    X[s+1] = ZeroExtend(data<datasize-1:0>, regsize);
else
    X[s] = ZeroExtend(data<datasize-1:0>, regsize);
    X[s+1] = ZeroExtend(data<2*datasize-1:datasize>, regsize);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

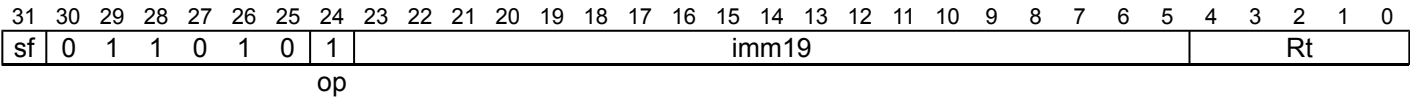
(old)

htmldiff from-

(new)

CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.



32-bit (sf == 0)

```
CBNZ <Wt>, <label>
```

64-bit (sf == 1)

```
CBNZ <Xt>, <label>
```

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- <label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(datasize) operand1 = X[t];

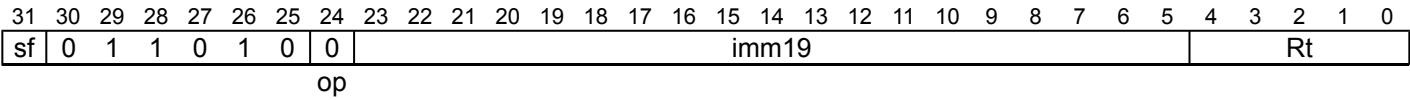
if IsZero(operand1) == iszero then
    BranchTo(PC[] + offset, BranchType_DIRBranchType_JMP);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



32-bit (sf == 0)

```
CBZ <Wt>, <label>
```

64-bit (sf == 1)

```
CBZ <Xt>, <label>
```

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- <label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(datasize) operand1 = X[t];

if IsZero(operand1) == iszero then
    BranchTo(PC[] + offset, BranchType_DIRBranchType_JMP);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CFINV

Invert Carry Flag. This instruction inverts the value of the PSTATE.C flag.

System  
(ARMv8.4)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	(0)	(0)	(0)	(0)	0	0	0	1	1	1	1	1
																					CRm										

System

CFINV

if !HaveFlagManipulateExt() then UNDEFINED; (~~) then UnallocatedEncoding();~~

Operation

PSTATE.C = NOT(PSTATE.C);

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CFP

Control Flow Prediction Restriction by Context prevents control flow predictions that predict execution addresses, based on information gathered from earlier execution within a particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see *CFP RCTX, Control Flow Prediction Restriction by Context*.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	1	1	1	0	0	1	1	1	0	0	Rt				
L										op1				CRn				CRm				op2									

## System

CFP RCTX, [<Xt>](#)

is equivalent to

[SYS](#) #3, C8, C3, #4, [<Xt>](#)

and is always the preferred disassembly.

## Assembler Symbols

[<Xt>](#) Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

## Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

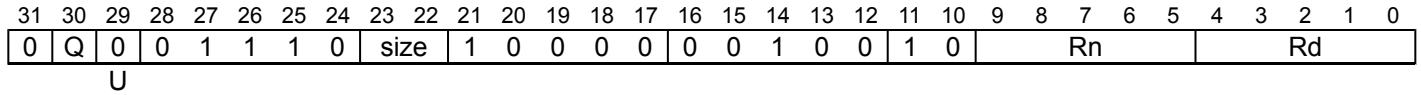
Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CLS (vector)

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The count does not include the most significant bit itself.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Vector

CLS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:



- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## CLS

~~Count leading sign bits:  $Rd = CLS(Rn)$ .~~

Count Leading Sign bits counts the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. This count does not include the most significant bit of the source register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1	Rn						Rd							
																						op													

### 32-bit (sf == 0)

CLS <Wd>, <Wn>

### 64-bit (sf == 1)

CLS <Xd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
CountOp opcode = if op == '0' then CountOp_CLZ else CountOp_CLS;
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
integer result;
bits(datasize) operand1 = X[n];

if opcode == CountOp_CLZ then
    result = CountLeadingZeroBits(operand1);
else
    result = CountLeadingSignBits(operand1);

X[d] = result<datasize-1:0>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CLZ (vector)

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	0	1	0	0	1	0	Rn				Rd					
U																															

### Vector

CLZ <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

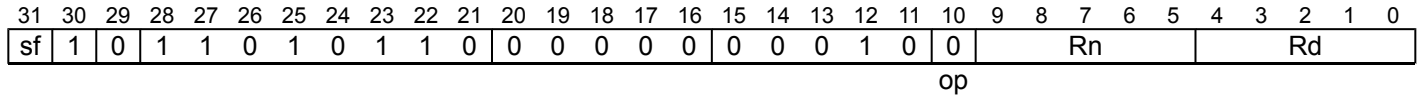
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## CLZ

~~Count leading zero bits: Rd = CLZ(Rn).~~

Count Leading Zeros counts the number of binary zero bits before the first binary one bit in the value of the source register, and writes the result to the destination register.



### 32-bit (sf == 0)

CLZ <Wd>, <Wn>

### 64-bit (sf == 1)

CLZ <Xd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
CountOp opcode = if op == '0' then CountOp_CLZ else CountOp_CLS;
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
integer result;
bits(datasize) operand1 = X[n];

if opcode == CountOp_CLZ then
    result = CountLeadingZeroBits(operand1);
else
    result = CountLeadingSignBits(operand1);

X[d] = result<datasize-1:0>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CMEQ (register)

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1				Rm			1	0	0	0	1	1										
U																															

### Scalar

CMEQ <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1				Rm			1	0	0	0	1	1										
U																															

### Vector

CMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;

```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## CMEQ (zero)

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	0	0	0	0	0	0	1	0	0	1	1	0	Rn				Rd					
U								op																							

### Scalar

CMEQ <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	1	0	0	1	1	0	Rn				Rd					
U								op																							

### Vector

CMEQ <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```



Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

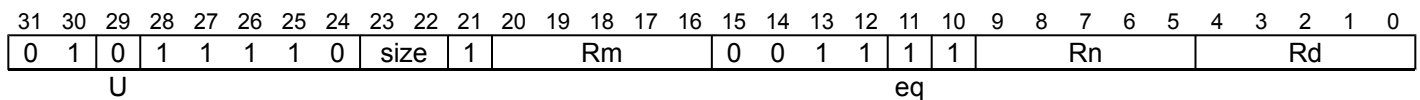
## CMGE (register)

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

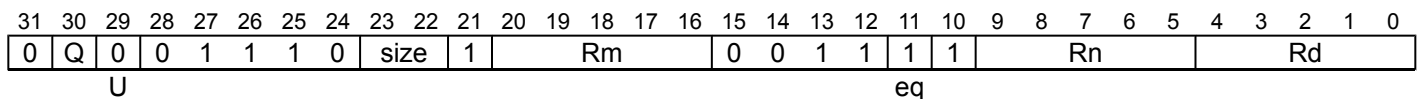


### Scalar

CMGE <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Vector



### Vector

CMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25**~~v29.05~~, AdvSIMD **v27.01**~~v26.0~~, pseudocode **v85-xml-00bet8\_rc3**~~v35.3~~; Build timestamp: **2018-09-13T13:20:09**~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## CMGE (zero)

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1	0	0	0	0	0	0	1	0	0	0	0	1	0	Rn				Rd				
U								op																							

### Scalar

CMGE <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	1	0	0	0	0	1	0	Rn				Rd				
U								op																							

### Vector

CMGE <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CMGT (register)

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1				Rm			0	0	1	1	0	1										
U								eq																							

### Scalar

CMGT <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			0	0	1	1	0	1										
U								eq																							

### Vector

CMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25**~~v29.05~~, AdvSIMD **v27.01**~~v26.0~~, pseudocode **v85-xml-00bet8\_rc3**~~v35.3~~; Build timestamp: **2018-09-13T13:20:09**~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## CMGT (zero)

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	0	0	0	0	0	0	1	0	0	0	0	1	0	Rn				Rd				
U								op																							

### Scalar

CMGT <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	1	0	0	0	0	1	0	Rn				Rd				
U								op																							

### Vector

CMGT <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```



Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CMHI (register)

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1				Rm			0	0	1	1	0	1										
U								eq																							

### Scalar

CMHI <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1				Rm			0	0	1	1	0	1										
U								eq																							

### Vector

CMHI <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25**~~v29.05~~, AdvSIMD **v27.01**~~v26.0~~, pseudocode **v85-xml-00bet8\_rc3**~~v35.3~~; Build timestamp: **2018-09-13T13:20:09**~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## CMHS (register)

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1				Rm			0	0	1	1	1	1										
U								eq																							

### Scalar

CMHS <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1				Rm			0	0	1	1	1	1										
U								eq																							

### Vector

CMHS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25**~~v29.05~~, AdvSIMD **v27.01**~~v26.0~~, pseudocode **v85-xml-00bet8\_rc3**~~v35.3~~; Build timestamp: **2018-09-13T13:20:09**~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## CMLE (zero)

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1	0	0	0	0	0	0	1	0	0	1	1	0	Rn				Rd					
U								op																							

### Scalar

CMLE <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	1	0	0	1	1	0	Rn				Rd					
U								op																							

### Vector

CMLE <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CMLT (zero)

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	0	0	0	0	0	0	1	0	1	0	1	0	Rn				Rd					

### Scalar

CMLT <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	1	0	1	0	1	0	Rn				Rd					

### Vector

CMLT <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.



- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;

```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

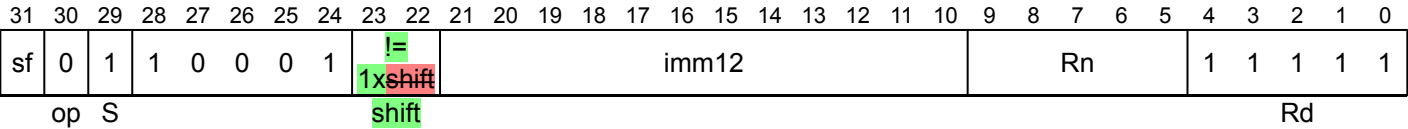
(old)	htmldiff from-	(new)
-------	----------------	-------

## CMN (immediate)

Compare Negative (immediate) adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This is an alias of [ADDS \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [ADDS \(immediate\)](#).
- The description of [ADDS \(immediate\)](#) gives the operational pseudocode for this instruction.



### 32-bit (sf == 0)

CMN <Wn|WSP>, #<imm>{, <shift>}

is equivalent to

ADDS WZR, <Wn|WSP>, #<imm> {, <shift>}

and is always the preferred disassembly.

### 64-bit (sf == 1)

CMN <Xn|SP>, #<imm>{, <shift>}

is equivalent to

ADDS XZR, <Xn|SP>, #<imm> {, <shift>}

and is always the preferred disassembly.

## Assembler Symbols

- <Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- <shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

shift<0>	<shift>
000	LSL #0
101	LSL #12
1x	RESERVED

## Operation

The description of [ADDS \(immediate\)](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+00:00~~2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

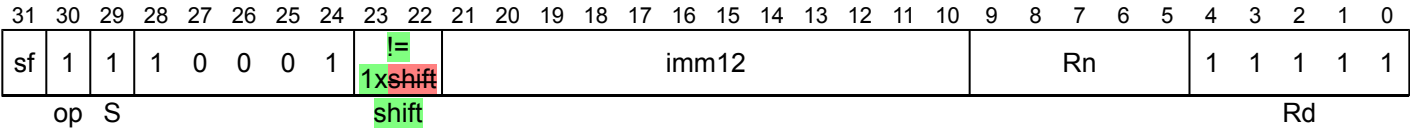
(old)	htmldiff from-	(new)
-------	----------------	-------

## CMP (immediate)

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of [SUBS \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [SUBS \(immediate\)](#).
- The description of [SUBS \(immediate\)](#) gives the operational pseudocode for this instruction.



### 32-bit (sf == 0)

CMP <Wn|WSP>, #<imm>{, <shift>}

is equivalent to

SUBS WZR, <Wn|WSP>, #<imm> {, <shift>}

and is always the preferred disassembly.

### 64-bit (sf == 1)

CMP <Xn|SP>, #<imm>{, <shift>}

is equivalent to

SUBS XZR, <Xn|SP>, #<imm> {, <shift>}

and is always the preferred disassembly.

## Assembler Symbols

- <Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- <shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

shift<0>	<shift>
000	LSL #0
101	LSL #12
1x	RESERVED

## Operation

The description of [SUBS \(immediate\)](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:20:04Z~~~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## CMPP

Compare with Tag subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

This is an alias of [SUBPS](#). This means:

- The encodings in this description are named to match the encodings of [SUBPS](#).
- The description of [SUBPS](#) gives the operational pseudocode for this instruction.

### Integer (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	1	1	1	0	1	0	1	1	0	Xm					0	0	0	0	0	0	Xn					1	1	1	1	1	Xd	

### Integer

CMPP <Xn|SP>, <Xm|SP>

is equivalent to

[SUBPS](#) XZR, <Xn|SP>, <Xm|SP>

and is always the preferred disassembly.

### Assembler Symbols

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

### Operation

The description of [SUBPS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CMTST

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1				Rm			1	0	0	0	1	1										
U																															

### Scalar

CMTST <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			1	0	0	0	1	1										
U																															

### Vector

CMTST <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

## Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d>

Is the number of the SIMD&FP destination register, in the “Rd” field.

- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;

```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



CNT

Population Count per byte. This instruction counts the number of bits that have a value of one in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	1	0	1	1	0	Rn				Rd					

Vector

```
CNT <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '00' then UNDEFINED;
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8; if size != '00' then ReservedValue();
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    count = BitCount(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

(old)	htmldiff from-	(new)
-------	----------------	-------

## CPP

Cache Prefetch Prediction Restriction by Context prevents cache allocation predictions, based on information gathered from earlier execution within a particular execution context, from allowing later speculative execution within that context to be observable through side-channels. For more information, see [CPP RCTX, Cache Prefetch Prediction Restriction by Context](#).

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	1	1	1	0	0	1	1	1	1	1	1	Rt			
L										op1				CRn				CRm				op2									

## System

CPP RCTX, [<Xt>](#)

is equivalent to

[SYS](#) #3, C8, C3, #7, [<Xt>](#)

and is always the preferred disassembly.

## Assembler Symbols

[<Xt>](#) Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

## Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In ARMv8-A, this is an OPTIONAL instruction, and in ARMv8.1 it is mandatory for all implementations to implement it.

[ID\\_AA64ISAR0\\_EL1](#).CRC32 indicates whether this instruction is supported.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm				0	1	0	0	sz	Rn				Rd							
C																															

### CRC32B (sf == 0 && sz == 00)

CRC32B <Wd>, <Wn>, <Wm>

### CRC32H (sf == 0 && sz == 01)

CRC32H <Wd>, <Wn>, <Wm>

### CRC32W (sf == 0 && sz == 10)

CRC32W <Wd>, <Wn>, <Wm>

### CRC32X (sf == 1 && sz == 11)

CRC32X <Wd>, <Wn>, <Xm>

```

if !HaveCRCEXT() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UnallocatedEncoding();
if sf == '0' && sz == '11' then UnallocatedEncoding(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
();
integer size = 8 << UInt(sz); // 2-bit size field -> 8, 16, 32, 64
boolean crc32c = (C == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
<Wm>	Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

## Operation

```
bits(32)      acc      = X[n];    // accumulator
bits(size)    val      = X[m];    // input value
bits(32)      poly     = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;

bits(32+size) tempacc = BitReverse(acc) : Zeros(size);
bits(size+32) tempval = BitReverse(val) : Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In ARMv8-A, this is an OPTIONAL instruction, and in ARMv8.1 it is mandatory for all implementations to implement it.

[ID\\_AA64ISAR0\\_EL1](#).CRC32 indicates whether this instruction is supported.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	0	1	sz	Rn					Rd					
C																															

### CRC32CB (sf == 0 && sz == 00)

CRC32CB <Wd>, <Wn>, <Wm>

### CRC32CH (sf == 0 && sz == 01)

CRC32CH <Wd>, <Wn>, <Wm>

### CRC32CW (sf == 0 && sz == 10)

CRC32CW <Wd>, <Wn>, <Wm>

### CRC32CX (sf == 1 && sz == 11)

CRC32CX <Wd>, <Wn>, <Xm>

```

if !HaveCRCExt() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UnallocatedEncoding();
if sf == '0' && sz == '11' then UnallocatedEncoding(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
();
integer size = 8 << UInt(sz); // 2-bit size field -> 8, 16, 32, 64
boolean crc32c = (C == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
<Wm>	Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

## Operation

```
bits(32)    acc    = X[n];    // accumulator
bits(size)  val    = X[m];    // input value
bits(32)    poly   = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;

bits(32+size) tempacc = BitReverse(acc) : Zeros(size);
bits(size+32) tempval = BitReverse(val) : Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

No instruction other than branch instructions appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE prediction state for any SVE instructions.

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1
																CRm								op2							

## System

CSDB

```

SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp NOP;
  when '0000 001' op = SystemHintOp YIELD;
  when '0000 010' op = SystemHintOp WFE;
  when '0000 011' op = SystemHintOp WFI;
  when '0000 100' op = SystemHintOp SEV;
  when '0000 101' op = SystemHintOp SEVL;
  when '0000 111'
    SEE "XPACLR1";
  when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp TSB;
  when '0010 100'
    op = SystemHintOp CSDB;
  when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0'
    op = otherwise SystemHintOp BTI;
    BTypeCompatible = BTypeCompatible BTI(op2<2:1>);
  otherwise EndOfInstruction(); // Instruction executes as NOP

```



## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## DC

Data Cache operation. For more information, see .

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	1	op1			0	1	1	1	CRm			op2			Rt					
L											CRn																				

## System

DC <dc\_op>, <Xt>

is equivalent to

SYS #<op1>, C7, <Cm>, #<op2>, <Xt>

and is the preferred disassembly when `SysOp (op1, '0111', CRm, op2) == Sys_DC`.

## Assembler Symbols

<dc\_op>

Is a DC instruction name, as listed for the DC system instruction group, encoded in “op1:CRm:op2”:

op1	CRm	op2	<dc_op>	Architectural Feature
000	0110	001	IVAC	-
000	0110	010	ISW	-
000	01101010	011010	IGVACCSW	ARMv8.5-MemTag
000	01101110	100010	IGSWCISW	ARMv8.5-MemTag
000011	01100100	101001	IGDVACZVA	ARMv8.5-MemTag
000011	01101010	110001	IGDSWCVAC	ARMv8.5-MemTag
000011	10101011	010001	CSWCVAU	-
000011	10101100	100001	CGSWCVAP	ARMv8.5-MemTagARMv8.2-DCCPoP
000011	10101110	110001	CGDSWCIVAC	ARMv8.5-MemTag
000	1110	010	CISW	-
000	1110	100	CIGSW	ARMv8.5-MemTag
000	1110	110	CIGDSW	ARMv8.5-MemTag
011	0100	001	ZVA	-
011	0100	011	GVA	ARMv8.5-MemTag
011	0100	100	GZVA	ARMv8.5-MemTag
011	1010	001	CVAC	-
011	1010	011	CGVAC	ARMv8.5-MemTag
011	1010	101	CGDVAC	ARMv8.5-MemTag
011	1011	001	CVAU	-
011	1100	001	CVAP	ARMv8.2-DCCPoP
011	1100	011	CGVAP	ARMv8.5-MemTag
011	1100	101	CGDVAP	ARMv8.5-MemTag
011	1101	001	CVADP	ARMv8.2-DCCVADP
011	1101	011	CGVADP	ARMv8.5-MemTag
011	1101	101	CGDVADP	ARMv8.5-MemTag
011	1110	001	CIVAC	-
011	1110	011	CIGVAC	ARMv8.5-MemTag
011	1110	101	CIGDVAC	ARMv8.5-MemTag

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:18-06:16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

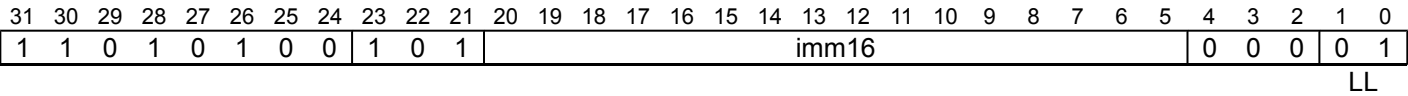
## DCPS1

- Debug Change PE State to EL1, when executed in Debug state:
- If executed at EL0 changes the current Exception level and SP to EL1 using SP\_EL1.
  - Otherwise, if executed at ELx, selects SP\_ELx.

- The target exception level of a DCPS1 instruction is:
- EL1 if the instruction is executed at EL0.
  - Otherwise, the Exception level at which the instruction is executed.

- When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:
- `ELR_ELx` becomes UNKNOWN.
  - `SPSR_ELx` becomes UNKNOWN.
  - `ESR_ELx` becomes UNKNOWN.
  - `DLR_EL0` and `DSPSR_EL0` become UNKNOWN.
  - The endianness is set according to `SCTLR_ELx.EE`.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and `HCR_EL2.TGE == 1`.  
This instruction is always UNDEFINED in Non-debug state.  
For more information on the operation of the DCPSn instructions, see [DCPS](#).



### System

```
DCPS1 {#<imm>}

bits(2) target_level = LL;
if LL == '00' then UNDEFINED;
if !if LL == '00' then UnallocatedEncoding();
if !Halted() then AArch64.UndefinedFault();
```

### Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

### Operation

```
DCPSInstruction(target_level);
```

Internal version only: isa `v30.25v29.05`, AdvSIMD `v27.01v26.0`, pseudocode `v85-xml-00bet8_rc3v35.3`; Build timestamp: `2018-09-13T13:2018-06-16T09:0445`

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## DCPS2

Debug Change PE State to EL2, when executed in Debug state:

- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP\_EL2.
- Otherwise, if executed at ELx, selects SP\_ELx.

The target exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

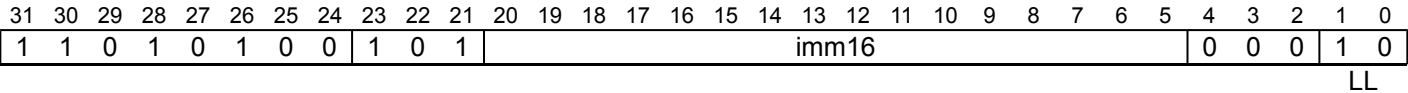
- *ELR\_ELx* becomes UNKNOWN.
- *SPSR\_ELx* becomes UNKNOWN.
- *ESR\_ELx* becomes UNKNOWN.
- *DLR\_EL0* and *DSPSR\_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR\_ELx*.EE.

This instruction is UNDEFINED at the following exception levels:

- All exception levels if EL2 is not implemented.
- At EL0 and EL1 ~~in Secure state~~ if EL2 is ~~disabled in the current Security state~~ ~~implemented~~.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see [DCPS](#).



### System

DCPS2 {#<imm>}

```
bits(2) target_level = LL;
if LL == '00' then UNDEFINED;
if !if LL == '00' then UnallocatedEncoding();
if !Halted() then AArch64.UndefinedFault();
```

### Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

### Operation

```
DCPSInstruction(target_level);
```

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13~~~~T13~~~~2018-06-16~~~~T09:04~~~~45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DCPS3

Debug Change PE State to EL3, when executed in Debug state:

- If executed at EL3 selects SP\_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP\_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

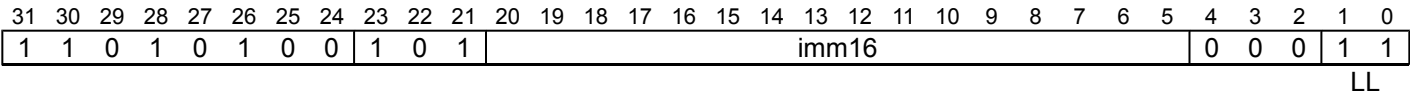
- *ELR\_EL3* becomes UNKNOWN.
- *SPSR\_EL3* becomes UNKNOWN.
- *ESR\_EL3* becomes UNKNOWN.
- *DLR\_EL0* and *DSPSR\_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR\_EL3*.EE.

This instruction is UNDEFINED at all exception levels if either:

- *EDSCR*.SDD == 1.
- EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS*.



System

```
DCPS3 {#<imm>}

bits(2) target_level = LL;
if LL == '00' then UNDEFINED;
if !if LL == '00' then UnallocatedEncoding();
if !Halted() then AArch64.UndefinedFault();
```

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
DCPSInstruction(target_level);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see [Data Memory Barrier](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm			1	0	1	1	1	1	1	1	1
																												opc			

## System

DMB <option>|<imm>

```

MemBarrierOp op;
MBarrierDomain domain;
MBarrierTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if otherwise HaveSBEExtUnallocatedEncoding() && CRm<3:0> == '0000' then
      op = ();
case CRm<3:2> of
  when '00' domain = MemBarrierOp_SB;
  else
    UNDEFINED;
case CRm<3:2> of
  when '00' domain = MBarrierDomain_OuterShareable;
  when '01' domain = MBarrierDomain_Nonshareable;
  when '10' domain = MBarrierDomain_InnerShareable;
  when '11' domain = MBarrierDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MBarrierTypes_Reads;
  when '10' types = MBarrierTypes_Writes;
  when '11' types = MBarrierTypes_All;
  otherwise
    if CRm<3:2> == '01' then
if CRm<3:2> == '00' then
      op = MemBarrierOp_PSSBB;
    elseif CRm<3:2> == '00' && opc == '00' then
      op = MemBarrierOp_SSBB;
    elseif elseif CRm<3:2> == '01' then
op = HaveSBEExtMemBarrierOp_PSSBB() && CRm<3:2> == '00' && opc == '11' then
      op = MemBarrierOp_SB;
    else
      types = MBarrierTypes_All;
      domain = MBarrierDomain_FullSystem;

```

## Assembler Symbols

<option> Specifies the limitation on the barrier operation. Values are:

### SY

Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

### ST

Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

## LD

Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

## ISH

Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

## ISHST

Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

## ISHL

Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

## NSH

Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

## NSHST

Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

## NSHL

Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

## OSH

Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

## OSHST

Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

## OSHL

Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see [Data Memory Barrier \(DMB\)](#) or see [Data Synchronization Barrier \(DSB\)](#).

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

## Operation

```
case op of
  when MemBarrierOp_DSBDataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier();
  when MemBarrierOp_SSBBSpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA();
  when (); MemBarrierOp_SBSpeculationBarrier();
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



DRPS

Debug restore process state.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0

System

DRPS

```
if !Halted() || PSTATE.EL == EL0 then UNDEFINED; then UnallocatedEncoding();
```

Operation

```
DRPSInstruction();
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see [Data Synchronization Barrier](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	!= 0x00			1	0	0	1	1	1	1	1						
																				CRm			opc													

## System

DSB <option>|<imm>

```

MemBarrierOp op;
MBarrierDomain domain;
MBarrierTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if otherwise HaveSBEExtUnallocatedEncoding() && CRm<3:0> == '0000' then
      op = ();
case CRm<3:2> of
  when '00' domain = MemBarrierOp_SB;
  else
    UNDEFINED;
case CRm<3:2> of
  when '00' domain = MBarrierDomain_OuterShareable;
  when '01' domain = MBarrierDomain_Nonshareable;
  when '10' domain = MBarrierDomain_InnerShareable;
  when '11' domain = MBarrierDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MBarrierTypes_Reads;
  when '10' types = MBarrierTypes_Writes;
  when '11' types = MBarrierTypes_All;
  otherwise
    if CRm<3:2> == '01' then
if CRm<3:2> == '00' then
      op = MemBarrierOp_PSSBB;
    elseif CRm<3:2> == '00' && opc == '00' then
      op = MemBarrierOp_SSBB;
    elseif elseif CRm<3:2> == '01' then
op = HaveSBEExtMemBarrierOp_PSSBB() && CRm<3:2> == '00' && opc == '11' then
      op = MemBarrierOp_SB;
    else
      types = MBarrierTypes_All;
      domain = MBarrierDomain_FullSystem;

```

## Assembler Symbols

<option> Specifies the limitation on the barrier operation. Values are:

### SY

Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

### ST

Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

## LD

Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

## ISH

Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

## ISHST

Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

## ISHLD

Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

## NSH

Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

## NSHST

Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

## NSHLD

Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

## OSH

Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

## OSHST

Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

## OSHLD

Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see [Data Memory Barrier \(DMB\)](#) or see [Data Synchronization Barrier \(DSB\)](#).

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

## Operation

```
case op of
  when MemBarrierOp_DSBDataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier();
  when MemBarrierOp_SSBBSpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA();
  when (); MemBarrierOp_SBSpeculationBarrier();
```

Internal version only: isa v30.25-v29.05, AdvSIMD v27.01-v26.0, pseudocode v85-xml-00bet8\_rc3-v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## DUP (element)

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [MOV \(scalar\)](#).

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	imm5					0	0	0	0	0	1	Rn				Rd					

### Scalar

DUP <V><d>, <Vn>.<T>[<index>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = if size > 3 then UnallocatedEncoding();
integer index = UInt(imm5<4:size+1>);
integer idxsize = if imm5<4> == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	imm5					0	0	0	0	0	1	Rn				Rd					

Vector

DUP <Vd>.<T>, <Vn>.<Ts>[<index>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = if size > 3 then UnallocatedEncoding();

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then ReservedValue(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then UNDEFINED;
();
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<T> For the scalar variant: is the element width specifier, encoded in “imm5”:

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

For the vector variant: is an arrangement specifier, encoded in “imm5:Q”:

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<V> Is the destination width specifier, encoded in “imm5”:

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxx1	imm5<4:1>
xx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(datasize) result;
bits(esize) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## DUP (general)

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	imm5					0	0	0	0	1	1	Rn					Rd				

### Advanced SIMD

DUP <Vd>.<T>, <R><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

// imm5<4:size+1> is IGNORED

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; if size > 3 then UnallocatedEncoding();

// imm5<4:size+1> is IGNORED

if size == 3 && Q == '0' then ReservedValue();
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "imm5:Q":

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<R> Is the width specifier for the general-purpose source register, encoded in "imm5":

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

Unspecified bits in "imm5" are ignored but should be set to zero by an assembler.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## DVP

Data Value Prediction Restriction by Context prevents data value predictions, based on information gathered from earlier execution within an particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see [DVP RCTX, Data Value Prediction Restriction by Context](#).

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	1	1	1	0	0	1	1	1	0	1	Rt				
										L			op1			CRn				CRm				op2							

## System

DVP RCTX, [<Xt>](#)

is equivalent to

[SYS](#) #3, C8, C3, #5, [<Xt>](#)

and is always the preferred disassembly.

## Assembler Symbols

[<Xt>](#) Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

## Operation

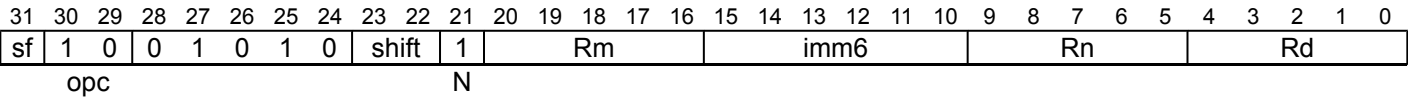
The description of [SYS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

### EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.



#### 32-bit (sf == 0)

```
EON <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

#### 64-bit (sf == 1)

```
EON <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then UNDEFINED;
ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

### Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

EOR3

Three-way Exclusive OR performs a three-way exclusive OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD  
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	0					Rm		0														Rd

Advanced SIMD

EOR3 <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

```
if !HaveSHA3Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt (Rd) ;
integer n = UInt (Rn) ;
integer m = UInt (Rm) ;

integer a = UInt (Ra) ;
```

Assembler Symbols

- <Vd>
- Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn>
- Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>
- Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Va>
- Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled() ;

bits(128) Vm = V[m] ;
bits(128) Vn = V[n] ;
bits(128) Va = V[a] ;
V[d] = Vn EOR Vm EOR Va ;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25-v29.05, AdvSIMD v27.01-v26.0, pseudocode v85-xml-00bet8\_rc3-v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## EOR (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sf		1		0		1		0		0		1		0		0		N		immr						imms						Rn						Rd			
opc																																									

### 32-bit (sf == 0 && N == 0)

EOR <Wd|WSP>, <Wn>, #<imm>

### 64-bit (sf == 1)

EOR <Xd|SP>, <Xn>, #<imm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Assembler Symbols

<Wd WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25~~v29.05~~, AdvSIMD v27.01~~v26.0~~, pseudocode v85-xml-00bet8\_rc3~~v35.3~~; Build timestamp: 2018-09-13T13~~2018-06-16T09~~:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

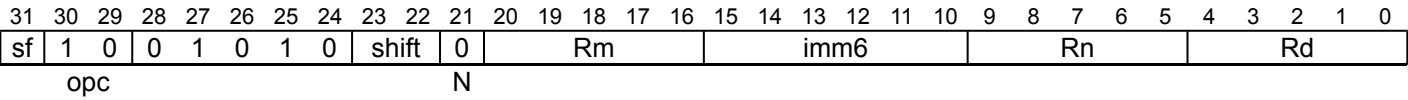
(old)

htmldiff from-

(new)

## EOR (shifted register)

Bitwise Exclusive OR (shifted register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.



### 32-bit (sf == 0)

```
EOR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
EOR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then UNDEFINED; if sf == '0' && imm6<5> == '1' then
ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

## Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06:16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores *PSTATE* from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERET is UNDEFINED at EL0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0									
																					A		M	Rn								op4								

System

ERET

```
if PSTATE.EL == EL0 then UNDEFINED;
boolean pac = (A == '1');
boolean use_key_a = (M == '0');

if !pac && op4 != '00000' then
    UNDEFINED;
elseif pac && (!thenUnallocatedEncoding());
boolean pac = (A == '1');
boolean use_key_a = (M == '0');

if !pac && op4 != '00000' then
    UnallocatedEncoding();
elseif pac && (!HavePACExt() || op4 != '11111') then
    UnallocatedEncoding();

if Rn != '11111' then
    UnallocatedEncoding() || op4 != '11111' then
        UNDEFINED;

if Rn != '11111' then
    UNDEFINED;();
```

Operation

```
AArch64.CheckForERetTrap(pac, use_key_a);
bits(64) target = ELR[];

if pac then
    if use_key_a then
        target = AuthIA(ELR[], SP[]);
    else
        target = AuthIB(ELR[], SP[]);

AArch64.ExceptionReturn(target, SPSR[]);
```

Internal version only: isa v30.25~~v29.05~~, AdvSIMD v27.01~~v26.0~~, pseudocode v85-xml-00bet8\_rc3~~v35.3~~; Build timestamp: 2018-09-13T13~~2018-06-16T09~~:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ERETAA, ERETAB

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores *PSTATE* from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for ERETAA, and key B is used for ERETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERETAA and ERETAB are UNDEFINED at EL0.

### Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1		
																					A						Rn						op4

### ERETAA (M == 0)

ERETAA

### ERETAB (M == 1)

ERETAB

```

if PSTATE.EL == EL0 then UNDEFINED;
boolean pac = (A == '1');
boolean use_key_a = (M == '0');

if !pac && op4 != '00000' then
    UNDEFINED;
elseif pac && (!then UnallocatedEncoding());
boolean pac = (A == '1');
boolean use_key_a = (M == '0');

if !pac && op4 != '00000' then
    UnallocatedEncoding();
elseif pac && (!HavePACExt() || op4 != '11111') then
    UnallocatedEncoding();

if Rn != '11111' then
    UnallocatedEncoding() || op4 != '11111' then
        UNDEFINED;

if Rn != '11111' then
    UNDEFINED;();

```

### Operation

```

AArch64.CheckForERetTrap(pac, use_key_a);
bits(64) target = ELR[];

if pac then
    if use_key_a then
        target = AuthIA(ELR[], SP[]);
    else
        target = AuthIB(ELR[], SP[]);

AArch64.ExceptionReturn(target, SPSR[]);

```

(old)

htmldiff from-

(new)

## ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR\_EL1 and VDISR\_EL2.

This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the ARM(R) Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for ARMv8-A architecture profile.

If the RAS Extension is not implemented, this instruction executes as a NOP.

### System (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	0	1	1	1	1	1
																CRm				op2											

### System

ESB

```

SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111'
    SEE "XPACLR1";
  when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 100'
    op = SystemHintOp_CSDB;
  when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0'
    op = otherwise SystemHintOp_BTI;
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction(); // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

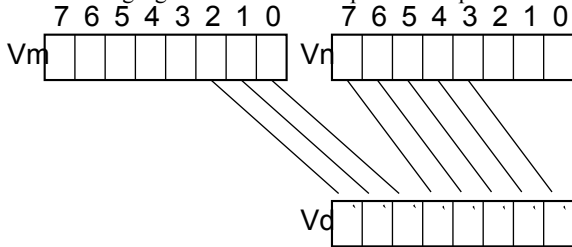
htmldiff from-

(new)

## EXT

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows an example of the operation of EXT doubleword operation for  $Q = 0$  and  $\text{imm4}<2:0> = 3$ .



Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	0	0	Rm				0	imm4				0	Rn				Rd						

### Advanced SIMD

EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if Q == '0' && imm4<3> == '1' then UNDEFINED;

integer datasize = if Q == '1' then 128 else 64;
integer position = if Q == '0' && imm4<3> == '1' then UnallocatedEncoding();

integer datasize = if Q == '1' then 128 else 64;
integer position = UInt(imm4) << 3;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the lowest numbered byte element to be extracted, encoded in "Q:imm4":

Q	imm4<3>	<index>
0	0	imm4<2:0>
0	1	RESERVED
1	x	imm4

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m];
bits(datasize) lo = V[n];
bits(datasize*2) concat = hi : lo;

V[d] = concat<position+datasize-1:position>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## EXTR

Extract register extracts a register from a pair of registers.

This instruction is used by the alias [ROR \(immediate\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	1	1	1	N	0	Rm					imms					Rn					Rd					

### 32-bit (sf == 0 && N == 0 && imms == 0xxxxx)

```
EXTR <Wd>, <Wn>, <Wm>, #<lsb>
```

### 64-bit (sf == 1 && N == 1)

```
EXTR <Xd>, <Xn>, <Xm>, #<lsb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
integer lsb;
```

```
if N != sf then UNDEFINED;
if sf == '0' && imms<5> == '1' then UNDEFINED;
lsb = if N != sf then UnallocatedEncoding();
if sf == '0' && imms<5> == '1' then ReservedValue();
lsb = UInt(imms);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<lsb>	For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">ROR (immediate)</a>	Rn == Rm



Operation

```
bits(datasize) result;  
bits(datasize) operand1 = X[n];  
bits(datasize) operand2 = X[m];  
bits(2*datasize) concat = operand1:operand2;  
  
result = concat<lsb+datasize-1:lsb>;  
  
X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FABD

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	0	Rm				0	0	0	1	0	1	Rn				Rd						

### Scalar half precision

FABD <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	Rm				1	1	0	1	0	1	Rn				Rd						

### Scalar single-precision and double-precision

FABD <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

### Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	0	Rm				0	0	0	1	0	1	Rn				Rd						

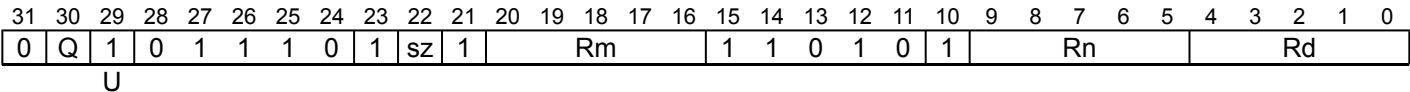
U

Vector half precision

FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) diff;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FABS (vector)

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	1	1	0	Rn				Rd					
U																															

### Half-precision

FABS <Vd>.<T>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	0	1	1	1	1	1	0	Rn				Rd				
U																															

### Single-precision and double-precision

FABS <Vd>.<T>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

sz	Q	<I>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
------	--

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FABS (scalar)

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	0	0	0	0	1	1	0	0	0	0	Rn				Rd					
																opc															

### Half-precision (type == 11) (ARMv8.2)

FABS <Hd>, <Hn>

### Single-precision (type == 00)

FABS <Sd>, <Sn>

### Double-precision (type == 01)

FABS <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
    when '11'
      if HaveFP16Ext() then
        datasize = 16;
      else
        UnallocatedEncoding() then
          datasize = 16;
        else
          UNDEFINED; ();

FPUnaryOp fpop;
case opc of
  when '00' fpop = FPUnaryOp_MOV;
  when '01' fpop = FPUnaryOp_ABS;
  when '10' fpop = FPUnaryOp_NEG;
  when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
  when FPUnaryOp_MOV    result = operand;
  when FPUnaryOp_ABS    result = FPAbs(operand);
  when FPUnaryOp_NEG    result = FPNeg(operand);
  when FPUnaryOp_SQRT   result = FPSqrt(operand, FPCR);

V[d] = result;
```

Internal version only: isa v30.25-v29.05, AdvSIMD v27.01-v26.0, pseudocode v85-xml-00bet8\_rc3-v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## FACGE

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	0	Rm				0	0	1	0	1	1	Rn				Rd						
U								E				ac																			

### Scalar half precision

FACGE <Hd>, <Hn>, <Hm>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();

```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		1	1	1	1	1	0	0	sz		1	Rm				1	1	1	0	1	1	Rn				Rd					
U								E				ac																			

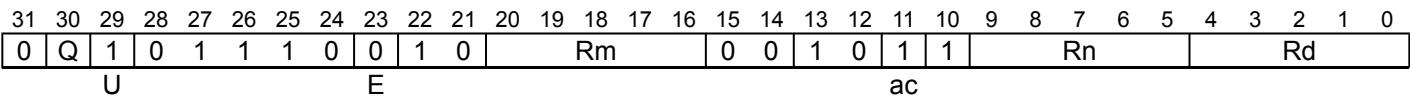
Scalar single-precision and double-precision

FACGE <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED; otherwise UnallocatedEncoding();
```

Vector half precision  
(ARMv8.2)



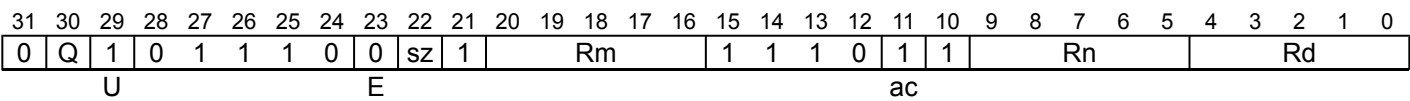
Vector half precision

FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED; ();
```

Vector single-precision and double-precision



Vector single-precision and double-precision

FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FACGT

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	0	Rm				0	0	1	0	1	1	Rn				Rd						
U								E				ac																			

### Scalar half precision

FACGT <Hd>, <Hn>, <Hm>

```

if !HaveFP16Ext() then UNDEFINED;

integer d =() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();

```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		1	1	1	1	1	0	1	sz		1	Rm				1	1	1	0	1	1	Rn				Rd					
U								E				ac																			

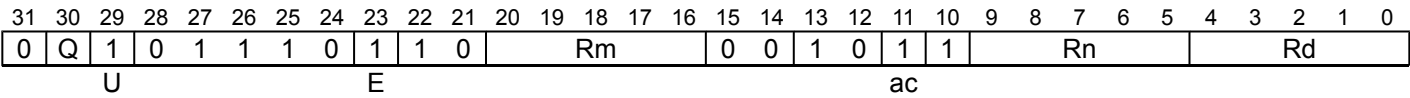
Scalar single-precision and double-precision

FACGT <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED; otherwise UnallocatedEncoding();
```

Vector half precision  
(ARMv8.2)



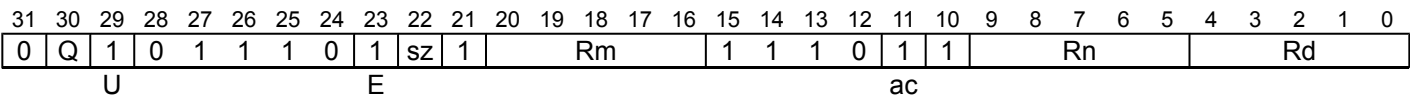
Vector half precision

FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED; ();
```

Vector single-precision and double-precision



Vector single-precision and double-precision

FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## FADD (vector)

Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm				0	0	0	1	0	1	Rn				Rd						
U																															

### Half-precision

```
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm					1	1	0	1	0	1	Rn					Rd				
U																															

### Single-precision and double-precision

```
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAAdd(element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

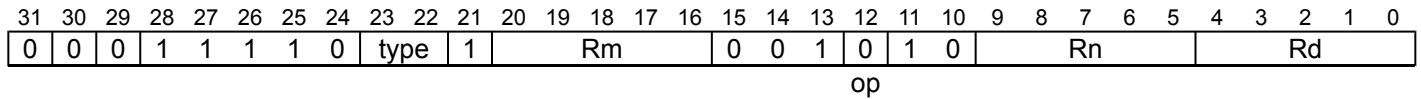
(new)

## FADD (scalar)

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Half-precision (type == 11) (ARMv8.2)

```
FADD <Hd>, <Hn>, <Hm>
```

### Single-precision (type == 00)

```
FADD <Sd>, <Sn>, <Sm>
```

### Double-precision (type == 01)

```
FADD <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding() then
        datasize = 16;
      else
        UNDEFINED;
  ();

boolean sub_op = (op == '1');
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

- <Sd>Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn>Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm>Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if sub_op then
    result = FPSub(operand1, operand2, FPCR);
else
    result = FPAdd(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FADDP (scalar)

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	1	1	0	Rn				Rd					
SZ																															

### Half-precision

FADDP <V><d>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
if sz == '1' then ReservedValue(Rn);
integer esize = 16;
if sz == '1' then UNDEFINED;
();
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = ReduceOp_FADD;

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	1	1	0	Rn				Rd					

### Single-precision and double-precision

FADDP <V><d>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = ReduceOp_FADD;

```

## Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2S
1	2D

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Internal version only: isa **v30.25v29.05**, AdvSIMD **v27.01v26.0**, pseudocode **v85-xml-00bet8\_rc3v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FADDP (vector)

Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm				0	0	0	1	0	1	Rn				Rd						
U																															

### Half-precision

FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm					1	1	0	1	0	1	Rn					Rd				
U																															

### Single-precision and double-precision

FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## FCADD

Floating-point Complex Add.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Three registers of the same type (ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	0				Rm			1	1	1	rot	0	1				Rn					Rd	

### Three registers of the same type

```
FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>
```

```
if !HaveFCADDExt() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << if size == '00' then ReservedValue();
if Q == '0' && size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then ReservedValue() && esize == 16 then UNDEFINED;
();
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate> Is the rotation, encoded in "rot":

rot	<rotate>
0	90
1	270

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2) -1
  case rot of
    when '0'
      element1 = FPNeg(Elem[operand2, e*2+1, esize]);
      element3 = Elem[operand2, e*2, esize];
    when '1'
      element1 = Elem[operand2, e*2+1, esize];
      element3 = FPNeg(Elem[operand2, e*2, esize]);
  Elem[result, e*2, esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR);
  Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCCMP

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the *PSTATE*. {N, Z, C, V} flags. If the condition does not pass then the *PSTATE*. {N, Z, C, V} flags are set to the flag bit specifier.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1						Rm				cond		0	1					Rn		0		nzcv
																														op	

### Half-precision (type == 11) (ARMv8.2)

```
FCCMP <Hn>, <Hm>, #<nzcv>, <cond>
```

### Single-precision (type == 00)

```
FCCMP <Sn>, <Sm>, #<nzcv>, <cond>
```

### Double-precision (type == 01)

```
FCCMP <Dn>, <Dm>, #<nzcv>, <cond>
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED;
        ();

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

## Assembler Symbols

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

#### NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(condition) then
    flags = FPCompare(operand1, operand2, signal_all_nans, FPCR);
PSTATE.<N,Z,C,V> = flags;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:16+09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

#### NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPCR* flags being set to N=0, Z=0, C=1, and V=1.

FCCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(condition) then
    flags = FPCompare(operand1, operand2, signal_all_nans, FPCR);
PSTATE.<N,Z,C,V> = flags;

```

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8~~rc3v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FCMEQ (register)

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	0	Rm				0	0	1	0	0	1	Rn				Rd						
U								E				ac																			

### Scalar half precision

FCMEQ <Hd>, <Hn>, <Hm>

```

if !HaveFP16Ext() then UNDEFINED;

integer d =() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();

```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0		1	0		1	1	1	1	0	0		sz		1	Rm				1	1	1	0	0		1	Rn				Rd			
U								E				ac																					

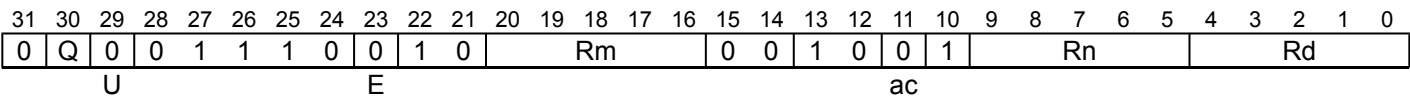
Scalar single-precision and double-precision

FCMEQ <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED; otherwise UnallocatedEncoding();
```

Vector half precision  
(ARMv8.2)



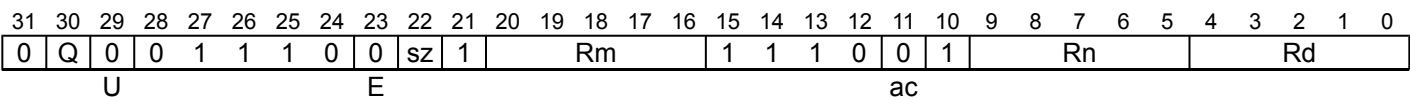
Vector half precision

FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();
```

Vector single-precision and double-precision





Vector single-precision and double-precision

FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FCMEQ (zero)

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn				Rd					
U										op																					

### Scalar half precision

FCMEQ <Hd>, <Hn>, #0.0

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn					Rd				
U										op																					

Scalar single-precision and double-precision

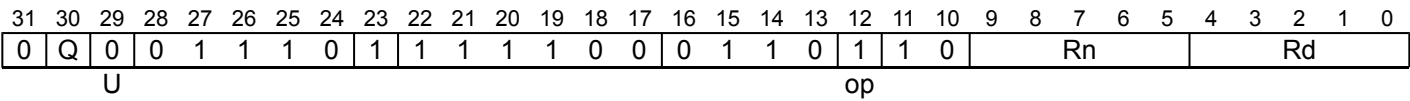
```
FCMEQ <V><d>, <V><n>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision
(ARMv8.2)



Vector half precision

```
FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0
```

```
if !HaveFP16Ext() then UNDEFINED;

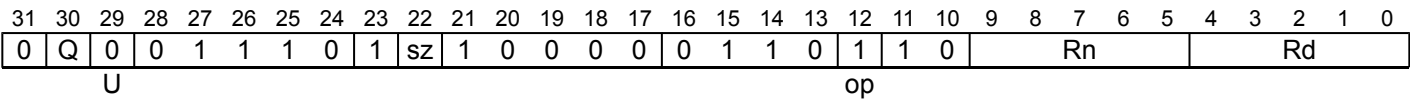
integer d =() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



## Vector single-precision and double-precision

FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

## Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
    when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
    when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FCMGE (register)

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	0	Rm				0	0	1	0	0	1	Rn				Rd						
U								E				ac																			

### Scalar half precision

FCMGE <Hd>, <Hn>, <Hm>

```

if !HaveFP16Ext() then UNDEFINED;

integer d =() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();

```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		1	1	1	1	1	0	0	sz		1	Rm				1	1	1	0	0		1	Rn				Rd				
U								E				ac																			

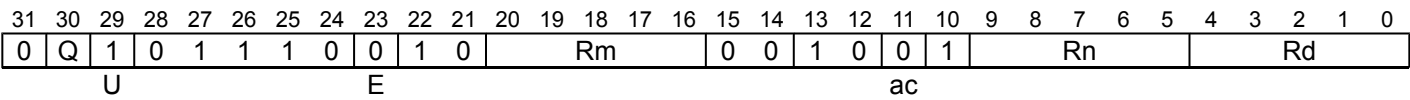
Scalar single-precision and double-precision

FCMGE <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED; otherwise UnallocatedEncoding();
```

Vector half precision  
(ARMv8.2)



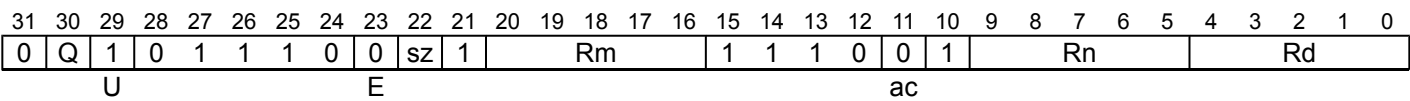
Vector half precision

FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED; ();
```

Vector single-precision and double-precision





Vector single-precision and double-precision

FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCMGE (zero)

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn						Rd					
U																op																	

### Scalar half precision

FCMGE <Hd>, <Hn>, #0.0

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn						Rd			
U																op															

Scalar single-precision and double-precision

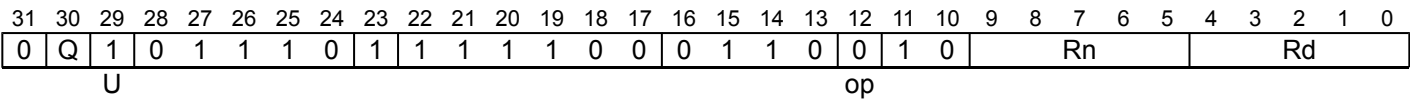
FCMGE <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision
(ARMv8.2)



Vector half precision

FCMGE <Vd>.<T>, <Vn>.<T>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

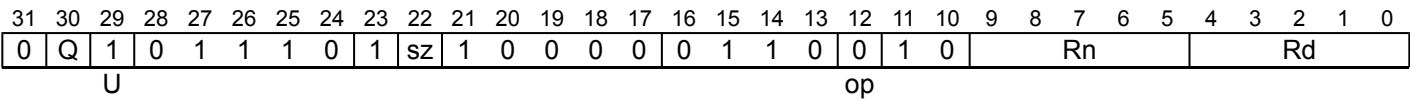
integer d =() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCMGE <Vd>.<T>, <Vn>.<T>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FCMGT (register)

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	0	Rm				0	0	1	0	0	1	Rn				Rd						
U								E				ac																			

### Scalar half precision

FCMGT <Hd>, <Hn>, <Hm>

```

if !HaveFP16Ext() then UNDEFINED;

integer d =() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();

```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		1	1	1	1	1	0	1	sz		1	Rm				1	1	1	0	0		1	Rn				Rd				
U								E				ac																			

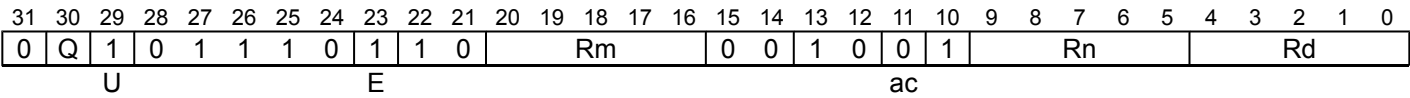
Scalar single-precision and double-precision

FCMGT <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED; otherwise UnallocatedEncoding();
```

Vector half precision  
(ARMv8.2)



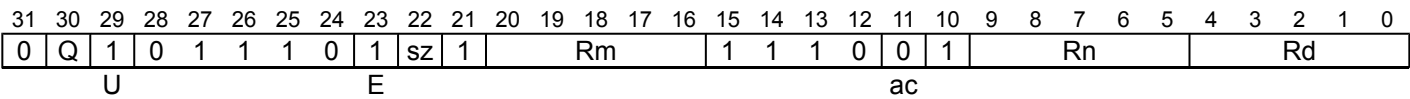
Vector half precision

FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED; ();
```

Vector single-precision and double-precision





Vector single-precision and double-precision

FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UnallocatedEncoding; abs = TRUE;
  otherwise UNDEFINED;();
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCMGT (zero)

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn				Rd					
U																op															

### Scalar half precision

FCMGT <Hd>, <Hn>, #0.0

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn						Rd							
U																op																			

Scalar single-precision and double-precision

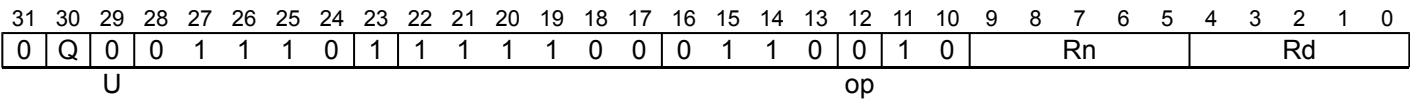
FCMGT <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision
(ARMv8.2)



Vector half precision

FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

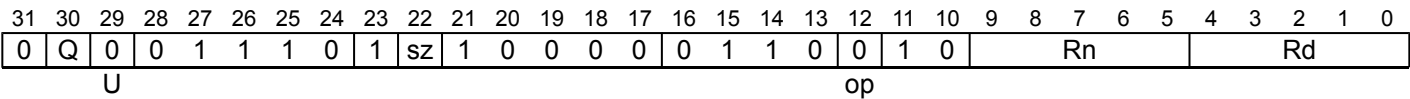
integer d =() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



Vector single-precision and double-precision

FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FCMLA (by element)

Floating-point Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Vector (ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size	L	M	Rm				0	rot	1	H	0	Rn				Rd							

(size == 01)

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

(size == 10)

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

```
if !HaveFCADDExt() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
if size == '00' || size == '11' then UNDEFINED;
if size == '01' then index = if size == '00' || size == '11' then ReservedValue();
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then ReservedValue();
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then ReservedValue();
if size == '01' && H == '1' && Q == '0' then ReservedValue() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then UNDEFINED;
if size == '01' && H == '1' && Q == '0' then UNDEFINED;();
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:H:L":

size	<index>
00	RESERVED
01	H:L
10	H
11	RESERVED

<rotate> Is the rotation, encoded in "rot":

rot	<rotate>
00	0
01	90
10	180
11	270

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to (elements DIV 2) -1
  case rot of
    when '00'
      element1 = Elem[operand2, index*2, esize];
      element2 = Elem[operand1, e*2, esize];
      element3 = Elem[operand2, index*2+1, esize];
      element4 = Elem[operand1, e*2, esize];
    when '01'
      element1 = FPNeg(Elem[operand2, index*2+1, esize]);
      element2 = Elem[operand1, e*2+1, esize];
      element3 = Elem[operand2, index*2, esize];
      element4 = Elem[operand1, e*2+1, esize];
    when '10'
      element1 = FPNeg(Elem[operand2, index*2, esize]);
      element2 = Elem[operand1, e*2, esize];
      element3 = FPNeg(Elem[operand2, index*2+1, esize]);
      element4 = Elem[operand1, e*2, esize];
    when '11'
      element1 = Elem[operand2, index*2+1, esize];
      element2 = Elem[operand1, e*2+1, esize];
      element3 = FPNeg(Elem[operand2, index*2, esize]);
      element4 = Elem[operand1, e*2+1, esize];

  Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, FPCR);
  Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);

V[d] = result;

```



(old)	htmldiff from-	(new)
-------	----------------	-------

## FCMLA

Floating-point Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Three registers of the same type

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	0				Rm			1	1	0	rot	1				Rn					Rd		

### Three registers of the same type

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

```

if !HaveFCADDExt() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << if size == '00' then ReservedValue();
if Q == '0' && size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then ReservedValue() && esize == 16 then UNDEFINED;
();
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

&lt;rotate&gt;

rot	<rotate>
00	0
01	90
10	180
11	270

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;

for e = 0 to (elements DIV 2) -1
    case rot of
        when '00'
            element1 = Elem[operand2, e*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, e*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, e*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, e*2, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, e*2+1, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, e*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];

        Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, FPCR);
        Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3 ; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FCMLE (zero)

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn				Rd					
U															op																

### Scalar half precision

FCMLE <Hd>, <Hn>, #0.0

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	0	0	1	1	0	1	1	0	Rn				Rd				
U										op																					

Scalar single-precision and double-precision

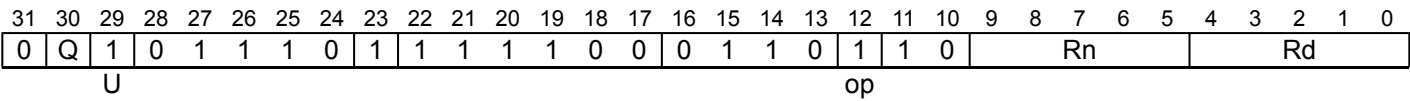
```
FCMLE <V><d>, <V><n>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision
(ARMv8.2)



Vector half precision

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

```
if !HaveFP16Ext() then UNDEFINED;

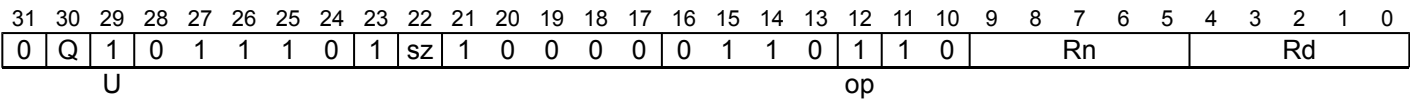
integer d =() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FCMLT (zero)

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn				Rd					

### Scalar half precision

FCMLT <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	0	1	0	Rn				Rd					

### Scalar single-precision and double-precision

FCMLT <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

### Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn				Rd					



Vector half precision

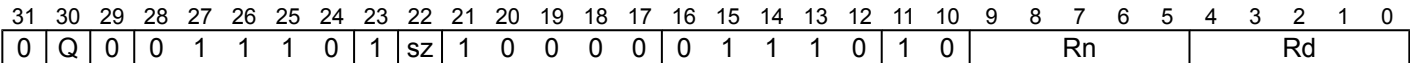
```
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
    when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
    when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

FCMP

Floating-point quiet Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the *PSTATE*. {N, Z, C, V} flags.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	1	1	1	0	type	1	Rm						0	0	1	0	0	0	Rn						0	x	0	0	0
opc																																

### Half-precision (type == 11 && opc == 00) (ARMv8.2)

FCMP <Hn>, <Hm>

### Half-precision, zero (type == 11 && Rm == (00000) && opc == 01) (ARMv8.2)

FCMP <Hn>, #0.0

### Single-precision (type == 00 && opc == 00)

FCMP <Sn>, <Sm>

### Single-precision, zero (type == 00 && Rm == (00000) && opc == 01)

FCMP <Sn>, #0.0

### Double-precision (type == 01 && opc == 00)

FCMP <Dn>, <Dm>

### Double-precision, zero (type == 01 && Rm == (00000) && opc == 01)

FCMP <Dn>, #0.0

```
integer n = UInt(Rn);
integer m = UInt(Rm);    // ignored when opc<0> == '1'

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED;
        ();

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

## Assembler Symbols

<Dn>	For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**<Sm>** Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of  $<$ ,  $=$ ,  $>$  or unordered. If either or both of the operands are NaNs, they are unordered, and all three of  $(\text{Operand1} < \text{Operand2})$ ,  $(\text{Operand1} = \text{Operand2})$  and  $(\text{Operand1} > \text{Operand2})$  are false. This case results in the *FPSCR* flags being set to  $N=0$ ,  $Z=0$ ,  $C=1$ , and  $V=1$ .

## Operation

```

CheckFPAdvSIMDEnabled64() ;

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = if cmp_with_zero then FPZero('0') else V[m];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:092018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

FCMPE

Floating-point signaling Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the *PSTATE*. {N, Z, C, V} flags.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1				Rm			0	0	1	0	0	0				Rn		1	x	0	0	0
																												opc			

### Half-precision (type == 11 && opc == 10) (ARMv8.2)

FCMPE <Hn>, <Hm>

### Half-precision, zero (type == 11 && Rm == (00000) && opc == 11) (ARMv8.2)

FCMPE <Hn>, #0.0

### Single-precision (type == 00 && opc == 10)

FCMPE <Sn>, <Sm>

### Single-precision, zero (type == 00 && Rm == (00000) && opc == 11)

FCMPE <Sn>, #0.0

### Double-precision (type == 01 && opc == 10)

FCMPE <Dn>, <Dm>

### Double-precision, zero (type == 01 && Rm == (00000) && opc == 11)

FCMPE <Dn>, #0.0

```
integer n = UInt(Rn);
integer m = UInt(Rm);    // ignored when opc<0> == '1'

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED;
        ();

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

## Assembler Symbols

<Dn>	For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

#### NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

FCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = if cmp_with_zero then FPZero('0') else V[m];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	Rm					cond					1	1	Rn					Rd				

### Half-precision (type == 11)

(ARMv8.2)

```
FCSEL <Hd>, <Hn>, <Hm>, <cond>
```

### Single-precision (type == 00)

```
FCSEL <Sd>, <Sn>, <Sm>, <cond>
```

### Double-precision (type == 01)

```
FCSEL <Dd>, <Dn>, <Dm>, <cond>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED;
        ();
bits(4) condition = cond;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;

result = if ConditionHolds(condition) then V[n] else V[m];

V[d] = result;

```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

FCVT

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the *FPCR* and writes the result to the SIMD&FP destination register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	0	1	opc	1	0	0	0	0	Rn				Rd							

### Half-precision to single-precision (type == 11 && opc == 00)

FCVT <Sd>, <Hn>

### Half-precision to double-precision (type == 11 && opc == 01)

FCVT <Dd>, <Hn>

### Single-precision to half-precision (type == 00 && opc == 11)

FCVT <Hd>, <Sn>

### Single-precision to double-precision (type == 00 && opc == 01)

FCVT <Dd>, <Sn>

### Double-precision to half-precision (type == 01 && opc == 11)

FCVT <Hd>, <Dn>

### Double-precision to single-precision (type == 01 && opc == 00)

FCVT <Sd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if type == opc then UNDEFINED;

integer srcsize;
case type of
  when '00' srcsize = 32;
  when '01' srcsize = 64;
  when '10' UNDEFINED;
  when '11' srcsize = 16;
integer dstsize;
case opc of
  when '00' dstsize = 32;
  when '01' dstsize = 64;
  when '10' UNDEFINED;
  when '11' dstsize = 16; if type == opc then UnallocatedEncoding();

integer sresize;
case type of
  when '00' sresize = 32;
  when '01' sresize = 64;
  when '10' UnallocatedEncoding();
  when '11' sresize = 16;
integer dstsize;
case opc of
  when '00' dstsize = 32;
  when '01' dstsize = 64;
  when '10' UnallocatedEncoding();
  when '11' dstsize = 16;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Hn>Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Dn>Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(dstsize) result;
bits(srcsize) operand = V[n];

result = FPConvert(operand, FPCR);
V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCVTAS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn						Rd					
U																																	

### Scalar half precision

FCVTAS <Hd>, <Hn>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn					Rd				
U																															

### Scalar single-precision and double-precision

FCVTAS <V><d>, <V><n>

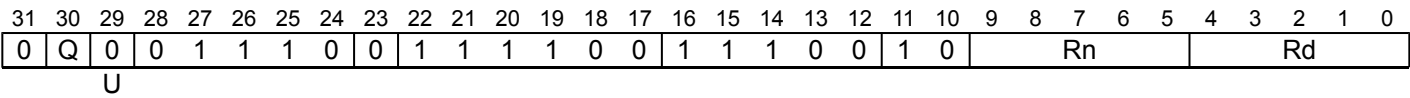
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector half precision  
(ARMv8.2)



Vector half precision

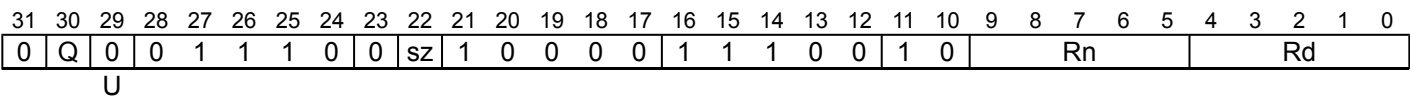
```
FCVTAS <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCVTAS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

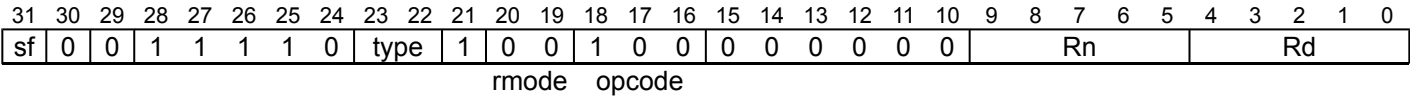


FCVTAS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
**(ARMv8.2)**

FCVTAS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
**(ARMv8.2)**

FCVTAS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTAS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTAS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTAS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTAS <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCVTAU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn				Rd					
U																															

### Scalar half precision

FCVTAU <Hd>, <Hn>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn						Rd					
U																																	

### Scalar single-precision and double-precision

FCVTAU <V><d>, <V><n>

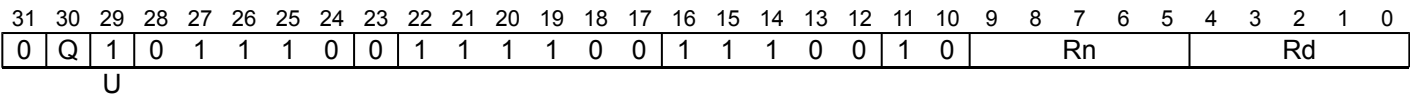
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector half precision  
(ARMv8.2)



Vector half precision

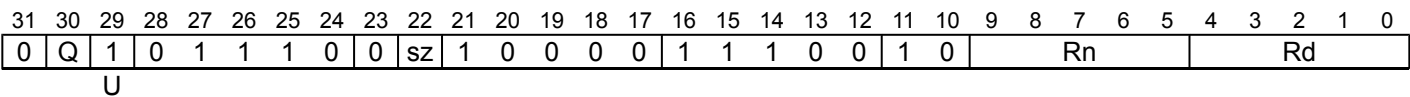
```
FCVTAU <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCVTAU <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

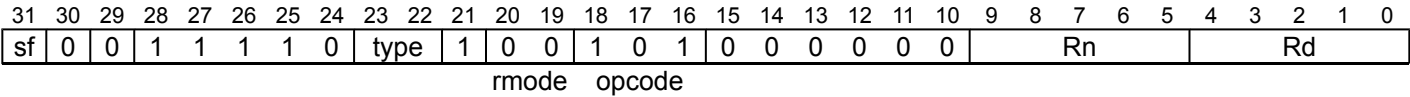
(new)

FCVTAU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.





**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTAU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTAU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTAU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTAU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTAU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTAU <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FCVTMS (vector)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2				o1																			

### Scalar half precision

FCVTMS <Hd>, <Hn>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2				o1																			

### Scalar single-precision and double-precision

FCVTMS <V><d>, <V><n>

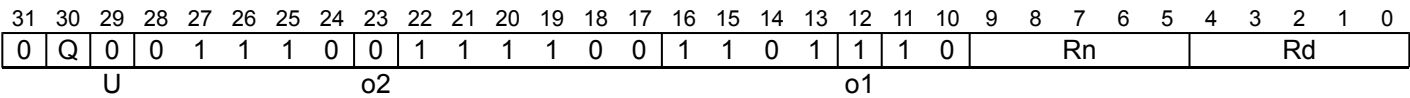
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision  
(ARMv8.2)



Vector half precision

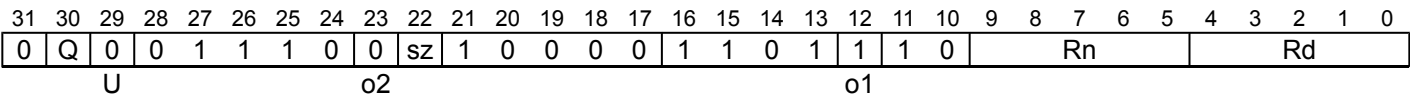
```
FCVTMS <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCVTMS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

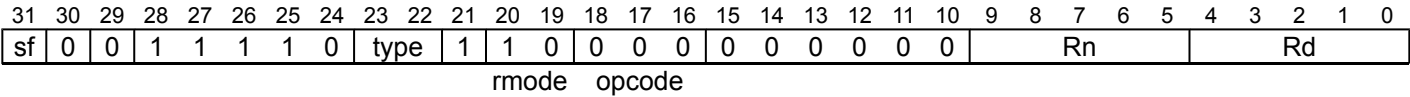
(new)

FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTMS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTMS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTMS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTMS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTMS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTMS <Xd>, <Dn>



```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64 ();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp\_CVT\_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp\_CVT\_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp\_MOV\_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp\_MOV\_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp\_CVT\_FtoI\_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

(new)

## FCVTMU (vector)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2				o1																			

### Scalar half precision

FCVTMU <Hd>, <Hn>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2				o1																			

### Scalar single-precision and double-precision

FCVTMU <V><d>, <V><n>

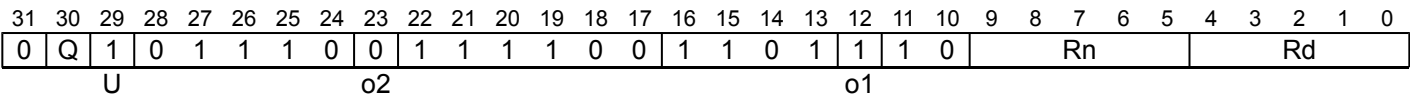
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision  
(ARMv8.2)



Vector half precision

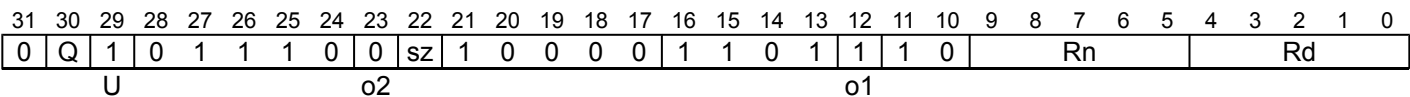
```
FCVTMU <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCVTMU <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D
- <d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n>

Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

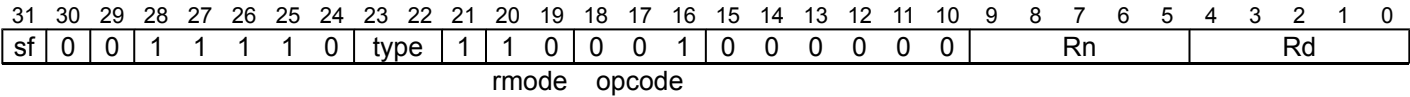
(new)

FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTMU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTMU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTMU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTMU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTMU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTMU <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.



<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCVTNS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn				Rd					
U								o2				o1																			

### Scalar half precision

FCVTNS <Hd>, <Hn>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn				Rd					
U								o2				o1																			

### Scalar single-precision and double-precision

FCVTNS <V><d>, <V><n>

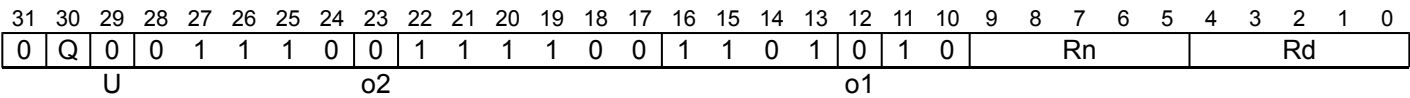
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision  
(ARMv8.2)



Vector half precision

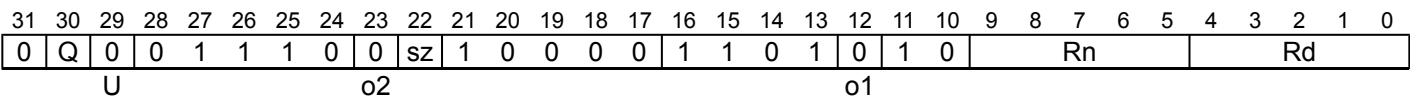
```
FCVTNS <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCVTNS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D
- <d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n>

Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

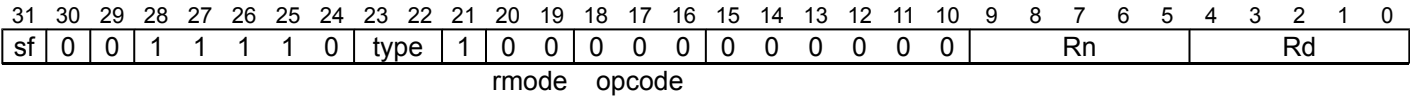
(new)

FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
**(ARMv8.2)**

FCVTNS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
**(ARMv8.2)**

FCVTNS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTNS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTNS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTNS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTNS <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## FCVTNU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn				Rd					
U								o2							o1																

### Scalar half precision

FCVTNU <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn				Rd					
U								o2							o1																

### Scalar single-precision and double-precision

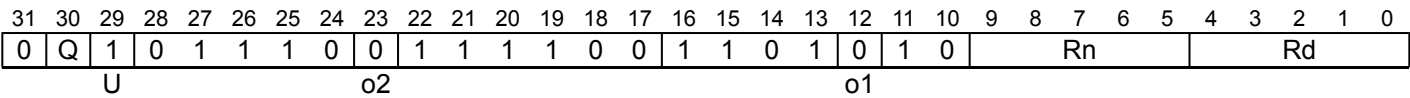
FCVTNU <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision  
(ARMv8.2)



Vector half precision

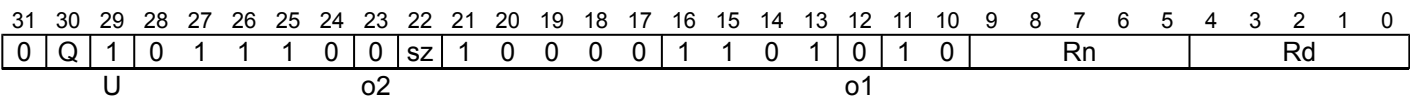
```
FCVTNU <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCVTNU <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

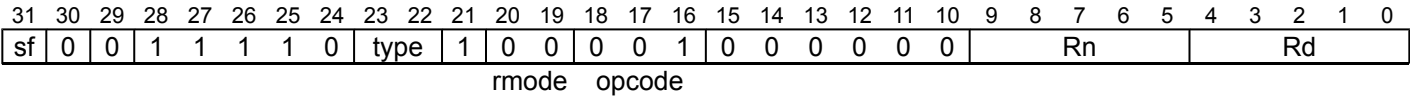
(new)

FCVTNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
**(ARMv8.2)**

FCVTNU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
**(ARMv8.2)**

FCVTNU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTNU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTNU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTNU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTNU <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCVTPS (vector)

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn					Rd				
U								o2							o1																

### Scalar half precision

FCVTPS <Hd>, <Hn>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
U								o2							o1																

### Scalar single-precision and double-precision

FCVTPS <V><d>, <V><n>

```

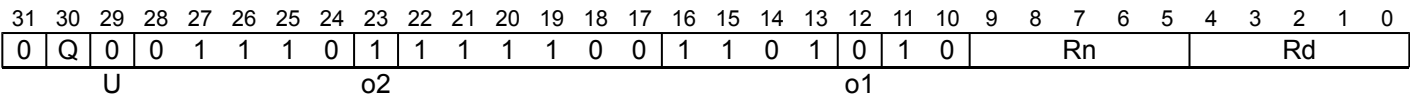
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```



Vector half precision  
(ARMv8.2)



Vector half precision

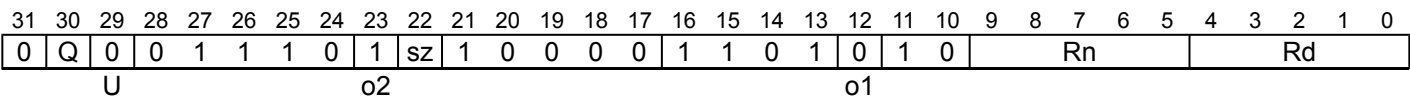
```
FCVTPS <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCVTPS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

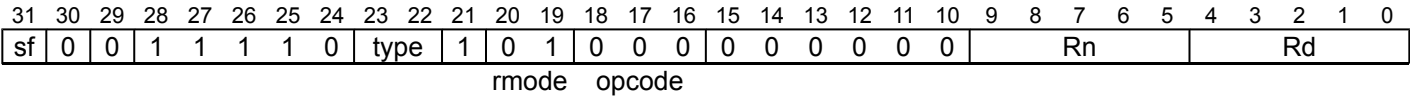
(new)

FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTPS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTPS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTPS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTPS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTPS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTPS <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FCVTPU (vector)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn						Rd					
U								o2							o1																		

### Scalar half precision

FCVTPU <Hd>, <Hn>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
U								o2							o1																

### Scalar single-precision and double-precision

FCVTPU <V><d>, <V><n>

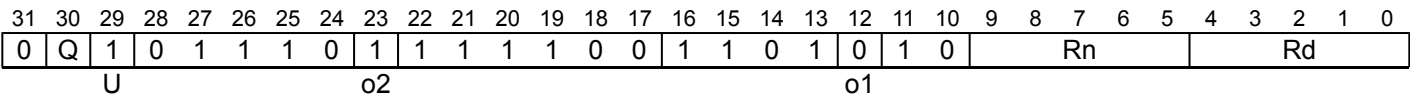
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision  
(ARMv8.2)



Vector half precision

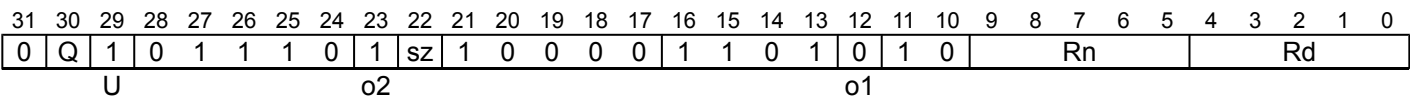
```
FCVTPU <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCVTPU <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D
- <d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n>

Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

For the vector half precision variant: is an arrangement specifier, encoded in “Q”:



Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

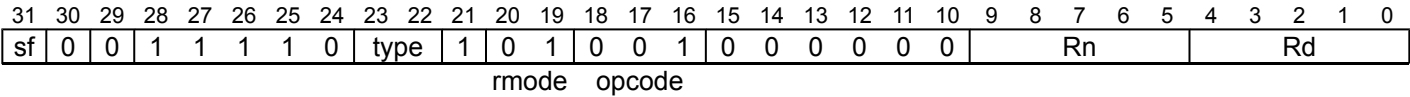
(new)

FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTPU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTPU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTPU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTPU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTPU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTPU <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCVTXN, FCVTXN2

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD&FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The FCVTXN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTXN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn				Rd					

### Scalar

FCVTXN <Vb><d>, <Va><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = esize;
integer elements = 1;
integer part = 0; if sz == '0' then ReservedValue();
integer esize = 32;
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn				Rd					

Vector

```
FCVTXN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = 64;
integer elements = 2;
integer part = if sz == '0' then ReservedValue();
integer esize = 32;
integer datasize = 64;
integer elements = 2;
integer part = UInt(Q);
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	x	RESERVED
1	0	2S
1	1	4S

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	RESERVED
1	2D

<Vb> Is the destination width specifier, encoded in “sz”:

sz	<Vb>
0	RESERVED
1	S

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “sz”:

sz	<Va>
0	RESERVED
1	D

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR, FPRounding_ODD);

Vpart[d, part] = result;
```

(old)	htmldiff from-	(new)
-------	----------------	-------



## FCVTZS (vector, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	Rn			Rd						
U									immh																						

### Scalar

FCVTZS <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - (()) then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	1	Rn				Rd								
U									immh																										

Vector

FCVTZS <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - (()) then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCVTZS (vector, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2							o1																

### Scalar half precision

FCVTZS <Hd>, <Hn>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2							o1																

### Scalar single-precision and double-precision

FCVTZS <V><d>, <V><n>

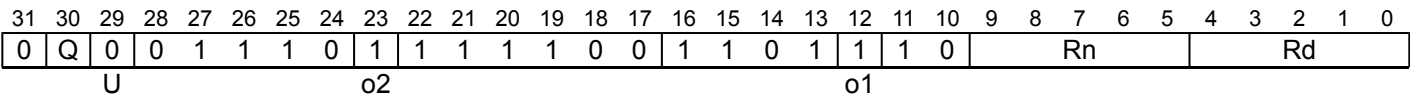
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision  
(ARMv8.2)



Vector half precision

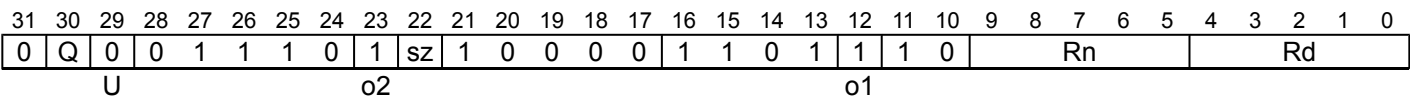
```
FCVTZS <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCVTZS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

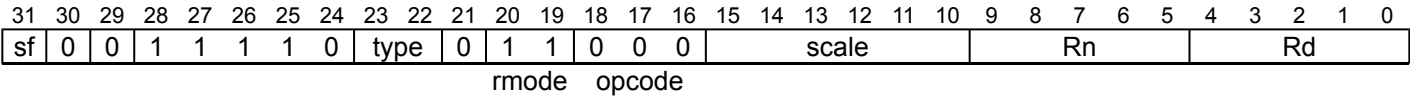
(new)

FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

```
FCVTZS <Wd>, <Hn>, #<fbits>
```

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

```
FCVTZS <Xd>, <Hn>, #<fbits>
```

**Single-precision to 32-bit (sf == 0 && type == 00)**

```
FCVTZS <Wd>, <Sn>, #<fbits>
```

**Single-precision to 64-bit (sf == 1 && type == 00)**

```
FCVTZS <Xd>, <Sn>, #<fbits>
```

**Double-precision to 32-bit (sf == 0 && type == 01)**

```
FCVTZS <Wd>, <Dn>, #<fbits>
```

**Double-precision to 64-bit (sf == 1 && type == 01)**

```
FCVTZS <Xd>, <Dn>, #<fbits>
```



```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                fltsize = 16;
            else
                UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - else UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
    when '00 11' // FCVTZ
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00' // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    otherwise
        UnallocatedEncoding;
    otherwise
        UNDEFINED; ();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	<p>For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".</p> <p>For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".</p>

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
    V[d] = fltval;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

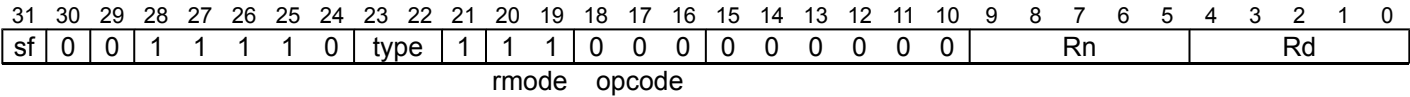
(old)	htmldiff from-	(new)
-------	----------------	-------

FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

FCVTZS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

FCVTZS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && type == 00)**

FCVTZS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && type == 00)**

FCVTZS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && type == 01)**

FCVTZS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && type == 01)**

FCVTZS <Xd>, <Dn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCVTZU (vector, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	1	Rn				Rd							
U									immh																									

### Scalar

FCVTZU <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - (()) then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	1	Rn				Rd							
U									immh																									

Vector

FCVTZU <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - (Q == '1') then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))



Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FCVTZU (vector, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2							o1																

### Scalar half precision

FCVTZU <Hd>, <Hn>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn					Rd				
U								o2							o1																

### Scalar single-precision and double-precision

FCVTZU <V><d>, <V><n>

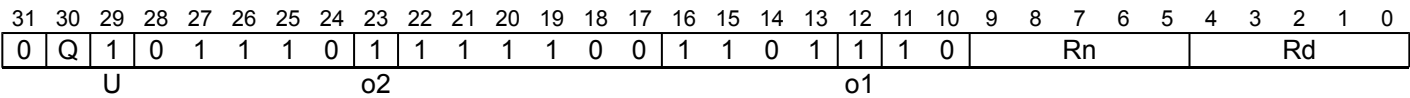
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision  
(ARMv8.2)



Vector half precision

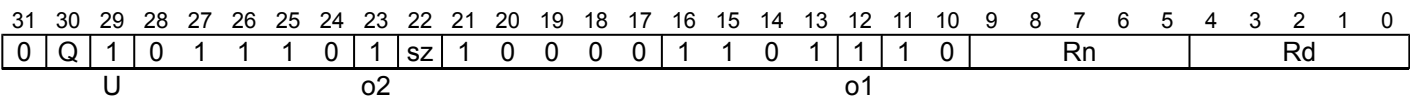
```
FCVTZU <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FCVTZU <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D
- <d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n>

Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sf		0		0		1		1		1		1		0		type		0		1		1		0		0		1		scale				Rn				Rd			
rmode												opcode																													

**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

```
FCVTZU <Wd>, <Hn>, #<fbits>
```

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

```
FCVTZU <Xd>, <Hn>, #<fbits>
```

**Single-precision to 32-bit (sf == 0 && type == 00)**

```
FCVTZU <Wd>, <Sn>, #<fbits>
```

**Single-precision to 64-bit (sf == 1 && type == 00)**

```
FCVTZU <Xd>, <Sn>, #<fbits>
```

**Double-precision to 32-bit (sf == 0 && type == 01)**

```
FCVTZU <Wd>, <Dn>, #<fbits>
```

**Double-precision to 64-bit (sf == 1 && type == 01)**

```
FCVTZU <Xd>, <Dn>, #<fbits>
```

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                fltsize = 16;
            else
                UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - else UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
    when '00 11' // FCVTZ
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00' // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    otherwise
        UnallocatedEncoding;
    otherwise
        UNDEFINED; ();

```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	<p>For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".</p> <p>For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".</p>

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
    V[d] = fltval;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

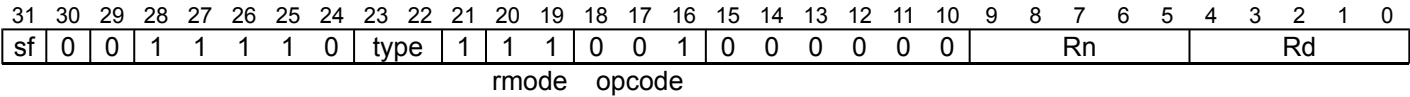


FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**Half-precision to 32-bit (sf == 0 && type == 11)**  
(ARMv8.2)

```
FCVTZU <Wd>, <Hn>
```

**Half-precision to 64-bit (sf == 1 && type == 11)**  
(ARMv8.2)

```
FCVTZU <Xd>, <Hn>
```

**Single-precision to 32-bit (sf == 0 && type == 00)**

```
FCVTZU <Wd>, <Sn>
```

**Single-precision to 64-bit (sf == 1 && type == 00)**

```
FCVTZU <Xd>, <Sn>
```

**Double-precision to 32-bit (sf == 0 && type == 01)**

```
FCVTZU <Wd>, <Dn>
```

**Double-precision to 64-bit (sf == 1 && type == 01)**

```
FCVTZU <Xd>, <Dn>
```

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

FDIV (vector)

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm				0	0	1	1	1	1	Rn				Rd						

Half-precision

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm					1	1	1	1	1	1	Rn					Rd				

Single-precision and double-precision

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<I>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FDIV (scalar)

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1				Rm			0	0	0	1	1	0				Rn				Rd		

### Half-precision (type == 11)

(ARMv8.2)

```
FDIV <Hd>, <Hn>, <Hm>
```

### Single-precision (type == 00)

```
FDIV <Sd>, <Sn>, <Sm>
```

### Double-precision (type == 01)

```
FDIV <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED; {}
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPDIV(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



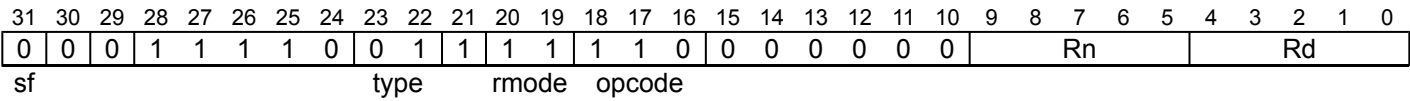
FJCVTZS

Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be accommodated as a signed 32-bit integer, then the result is the integer modulo  $2^{32}$ , as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Double-precision to 32-bit  
(ARMv8.3)



## Double-precision to 32-bit

FJCVTZS <Wd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();
```

Assembler Symbols

- <Wd>Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Dn>Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

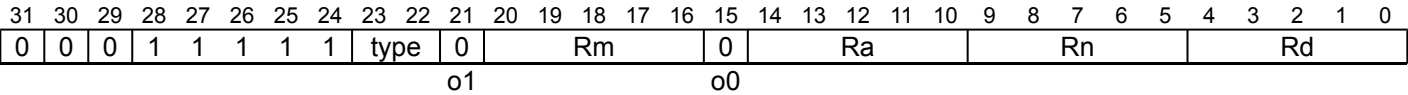
(old)	htmldiff from-	(new)
-------	----------------	-------

## FMADD

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Half-precision (type == 11) (ARMv8.2)

```
FMADD <Hd>, <Hn>, <Hm>, <Ha>
```

### Single-precision (type == 00)

```
FMADD <Sd>, <Sn>, <Sm>, <Sa>
```

### Double-precision (type == 01)

```
FMADD <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding() then
        datasize = 16;
    else
      UNDEFINED;
  ();

boolean opa_neg = (o1 == '1');
boolean opl_neg = (o0 != o1);
```

### Assembler Symbols

- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn>

Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- <Dm>

Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- <Da>

Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- <Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if opa_neg then operanda = FPNeg(operanda);
if opl_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FMAX (vector)

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm				0	0	1	1	0	1	Rn				Rd						
U								o1																							

### Half-precision

FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm					1	1	1	1	0	1	Rn					Rd				
U								o1																							

### Single-precision and double-precision

FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```

## FMAX (scalar)

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	Rm				0	1	0	0	1	0	Rn				Rd							
op																															

### Half-precision (type == 11) (ARMv8.2)

FMAX <Hd>, <Hn>, <Hm>

### Single-precision (type == 00)

FMAX <Sd>, <Sn>, <Sm>

### Double-precision (type == 01)

FMAX <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED; {}

FPMaXMinOp operation;
case op of
    when '00' operation = FPMaXMinOp_MAX;
    when '01' operation = FPMaXMinOp_MIN;
    when '10' operation = FPMaXMinOp_MAXNUM;
    when '11' operation = FPMaXMinOp_MINNUM;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.



<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
  when FPMaxMinOp_MAX      result = FPMax(operand1, operand2, FPCR);
  when FPMaxMinOp_MIN      result = FPMin(operand1, operand2, FPCR);
  when FPMaxMinOp_MAXNUM   result = FPMaxNum(operand1, operand2, FPCR);
  when FPMaxMinOp_MINNUM   result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FMAXNM (vector)

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm				0	0	0	0	0	1	Rn				Rd						
U								a																							

### Half-precision

FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm				1	1	0	0	0	1	Rn				Rd						
U								o1																							

### Single-precision and double-precision

FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaXNum(element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1				Rm			0	1	1	0	1	0				Rn				Rd		

op

### Half-precision (type == 11) (ARMv8.2)

```
FMAXNM <Hd>, <Hn>, <Hm>
```

### Single-precision (type == 00)

```
FMAXNM <Sd>, <Sn>, <Sm>
```

### Double-precision (type == 01)

```
FMAXNM <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED;{};

FPMaXMinOp operation;
case op of
    when '00' operation = FPMaXMinOp_MAX;
    when '01' operation = FPMaXMinOp_MIN;
    when '10' operation = FPMaXMinOp_MAXNUM;
    when '11' operation = FPMaXMinOp_MINNUM;
```

## Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
  when FPMaxMinOp_MAX      result = FPMax(operand1, operand2, FPCR);
  when FPMaxMinOp_MIN      result = FPMin(operand1, operand2, FPCR);
  when FPMaxMinOp_MAXNUM   result = FPMaxNum(operand1, operand2, FPCR);
  when FPMaxMinOp_MINNUM   result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1 sz																															

### Half-precision

FMAXNMP <V><d>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
if sz == '1' then ReservedValue(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
();
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMAXNMP <V><d>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;

```

## Assembler Symbols

**<V>** For the half-precision variant: is the destination width specifier, encoded in “sz”:

SZ	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

SZ	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

SZ	$\langle T \rangle$
0	2S
1	2D

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:09Z2018-06-16T09:04:45Z

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm				0	0	0	0	0	1	Rn				Rd						
U								a																							

### Half-precision

FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm					1	1	0	0	0	1	Rn					Rd				
U								o1																							

### Single-precision and double-precision

FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```



## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaXNum(element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
								o1																							

### Half-precision

FMAXNMV <V><d>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
								o1																							

### Single-precision and double-precision

FMAXNMV <V><d>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED; // .4S only
integer esize = 32 << if sz:Q != '01' then ReservedValue(); // .4S only
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;
```

## Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.  
For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

SZ	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:18+00:002018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
o1 sz																															

### Half-precision

FMAXP <V><d>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
if sz == '1' then ReservedValue(Rn);
integer esize = 16;
if sz == '1' then UNDEFINED;
();
integer datasize = esize * 2;
integer elements = 2;
ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
o1																															

### Single-precision and double-precision

FMAXP <V><d>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

```

## Assembler Symbols

**<V>** For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

SZ	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>SZ</b>	<b>&lt;T&gt;</b>
0	2S
1	2D

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3 ; Build timestamp: 2018-09-13T13:20:09Z2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMAXP (vector)

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm				0	0	1	1	0	1	Rn				Rd						
U								o1																							

### Half-precision

FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm				1	1	1	1	0	1	Rn				Rd						
U								o1																							

### Single-precision and double-precision

FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMIn(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaX(element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn				Rd					
o1																															

### Half-precision

FMAXV <V><d>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMAXV <V><d>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;
integer esize = 32 << if sz:Q != '01' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

```



## Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.  
For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

SZ	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMIN (vector)

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm				0	0	1	1	0	1	Rn				Rd						
U								o1																							

### Half-precision

FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm					1	1	1	1	0	1	Rn					Rd				
U								o1																							

### Single-precision and double-precision

FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
------	--

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMIn(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMMax(element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3 ; Build timestamp: 2018-09-13T13:20:092018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

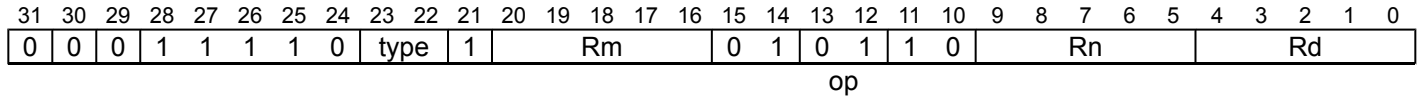
(new)

## FMIN (scalar)

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Half-precision (type == 11) (ARMv8.2)

FMIN <Hd>, <Hn>, <Hm>

### Single-precision (type == 00)

FMIN <Sd>, <Sn>, <Sm>

### Double-precision (type == 01)

FMIN <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
    when '11'
      if HaveFP16Ext() then
        datasize = 16;
      else
        UnallocatedEncoding() then
          datasize = 16;
        else
          UNDEFINED; {}

FPMaXMinOp operation;
case op of
  when '00' operation = FPMaXMinOp_MAX;
  when '01' operation = FPMaXMinOp_MIN;
  when '10' operation = FPMaXMinOp_MAXNUM;
  when '11' operation = FPMaXMinOp_MINNUM;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
  when FPMaxMinOp_MAX      result = FPMax(operand1, operand2, FPCR);
  when FPMaxMinOp_MIN      result = FPMin(operand1, operand2, FPCR);
  when FPMaxMinOp_MAXNUM   result = FPMaxNum(operand1, operand2, FPCR);
  when FPMaxMinOp_MINNUM   result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FMINNM (vector)

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm					0	0	0	0	0	1	Rn					Rd				
U								a																							

### Half-precision

FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm				1	1	0	0	0	1	Rn					Rd					
U								o1																							

### Single-precision and double-precision

FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

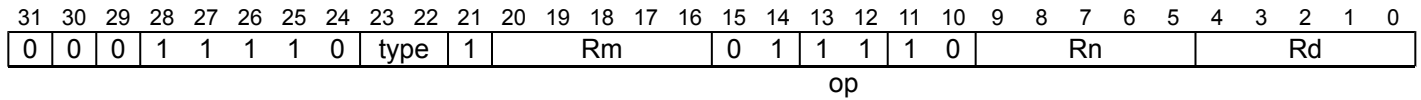
## FMINNM (scalar)

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Half-precision (type == 11) (ARMv8.2)

```
FMINNM <Hd>, <Hn>, <Hm>
```

### Single-precision (type == 00)

```
FMINNM <Sd>, <Sn>, <Sm>
```

### Double-precision (type == 01)

```
FMINNM <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED;{};

FPMMaxMinOp operation;
case op of
    when '00' operation = FPMMaxMinOp_MAX;
    when '01' operation = FPMMaxMinOp_MIN;
    when '10' operation = FPMMaxMinOp_MAXNUM;
    when '11' operation = FPMMaxMinOp_MINNUM;
```

## Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.



<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

case operation of
  when FPMaxMinOp_MAX      result = FPMax(operand1, operand2, FPCR);
  when FPMaxMinOp_MIN      result = FPMin(operand1, operand2, FPCR);
  when FPMaxMinOp_MAXNUM   result = FPMaxNum(operand1, operand2, FPCR);
  when FPMaxMinOp_MINNUM   result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
o1 sz																															

### Half-precision

FMINNMP <V><d>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
if sz == '1' then ReservedValue(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
();
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
o1																															

### Single-precision and double-precision

FMINNMP <V><d>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;

```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz		<V>
0		H
1		RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz		<V>
0		S
1		D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz		<T>
0		2H
1		RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz		<T>
0		2S
1		2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	0	Rm				0	0	0	0	0	1	Rn				Rd						
U								a																							

### Half-precision

FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm					1	1	0	0	0	1	Rn					Rd				
U								o1																							

### Single-precision and double-precision

FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
------	--

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaXNum(element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

(new)

## FMINNMV

Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

### Half-precision

FMINNMV <V><d>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMINNMV <V><d>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED; // .4S only
integer esize = 32 << if sz:Q != '01' then ReservedValue(); // .4S only
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMAXNUM;

```

## Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.  
For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

SZ	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:04Z2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

(new)

## FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
o1 sz																															

### Half-precision

FMINP <V><d>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
if sz == '1' then ReservedValue(Rn);
integer esize = 16;
if sz == '1' then UNDEFINED;
();
integer datasize = esize * 2;
integer elements = 2;
ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
o1																															

### Single-precision and double-precision

FMINP <V><d>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
integer elements = 2;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

```



## Assembler Symbols

**<V>** For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

SZ	<T>
0	2S
1	2D

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:00~~2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMINP (vector)

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	0	Rm				0	0	1	1	0	1	Rn				Rd						
U								o1																							

### Half-precision

FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm				1	1	1	1	0	1	Rn				Rd						
U								o1																							

### Single-precision and double-precision

FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMIn(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMMax(element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMINV

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn				Rd					
o1																															

### Half-precision

FMINV <V><d>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn				Rd					
o1																															

### Single-precision and double-precision

FMINV <V><d>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;
integer esize = 32 << if sz:Q != '01' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = if o1 == '1' then ReduceOp_FMIN else ReduceOp_FMAX;

```

## Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.  
For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

SZ	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	$\langle T \rangle$
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(op, operand, esize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:18Z2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMLA (by element)

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar, half-precision](#) , [Scalar, single-precision and double-precision](#) , [Vector, half-precision](#) and [Vector, single-precision and double-precision](#)

### Scalar, half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	0	L	M	Rm			0	0	0	1	H	0	Rn				Rd						
o2																															

### Scalar, half-precision

```
FMLA <Hd>, <Hn>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UNDEFINED;
integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = () then UnallocatedEncoding();
integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

### Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm				0	0	0	1	H	0	Rn				Rd					
																o2															

Scalar, single-precision and double-precision

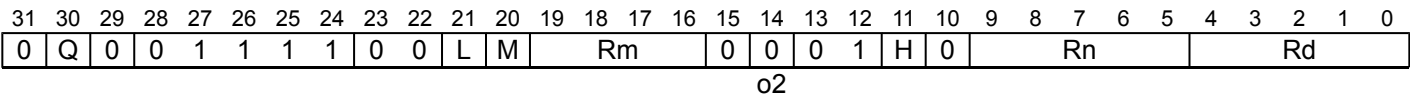
```
FMLA <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = when '11' UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Vector, half-precision
(ARMv8.2)



Vector, half-precision

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UNDEFINED;

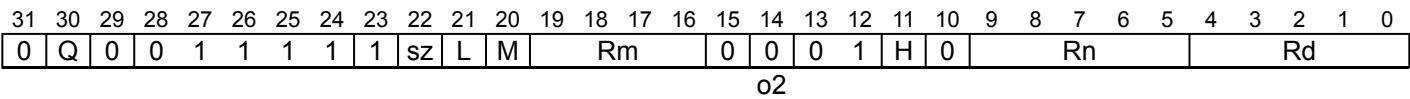
integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = () then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Vector, single-precision and double-precision



Vector, single-precision and double-precision

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
();

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.  
For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- <Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

- <index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.



For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FMLA (vector)

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm				0	0	0	0	1	1	Rn				Rd						
a																															

### Half-precision

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm					1	1	0	0	1	1	Rn					Rd				
op																															

### Single-precision and double-precision

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```

## Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<i>sz</i>	<b>Q</b>	<b>&lt;T&gt;</b>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

**<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
------	---

## Operation

```

checkFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:092018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

(new)

## FMLAL, FMLAL2 (by element)

Floating-point fused Multiply-Add Long to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

*ID\_AA64ISAR0\_EL1*.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: [FMLAL](#) and [FMLAL2](#)

### FMLAL

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	1	0	sz	L	M	Rm			0	0	0	0	H	0	Rn				Rd					
									sz	S																					

### FMLAL

FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.[<index>]

```

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then ReservedValue('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
();
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q=='1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;

```

### FMLAL2

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	1	0	sz	L	M	Rm			1	0	0	0	H	0	Rn				Rd					
									sz	S																					

FMLAL2

FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then ReservedValue('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
();
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q=='1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the element index, encoded in the "H:L:M" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n,part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMLAL, FMLAL2 (vector)

Floating-point fused Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

*ID\_AA64ISAR0\_EL1.FHM* indicates whether this instruction is supported.

It has encodings from 2 classes: [FMLAL](#) and [FMLAL2](#)

### FMLAL

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0sz	1	Rm				1	1	1	0	1	1	Rn				Rd						
									S	sz																					

### FMLAL

FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then ReservedValue(Rm);
if sz == '1' then UNDEFINED;
();
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;

```

### FMLAL2

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	0sz	1	Rm				1	1	0	0	1	1	Rn				Rd						
								S	sz																						

## FMLAL2

FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then ReservedValue(Rm);
if sz == '1' then UNDEFINED;
();
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;

```

## Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
------	---

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
------	--

<Tb>	Is an arrangement specifier, encoded in “Q”:
------	--

Q	<Tb>
0	2H
1	4H

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n,part];
bits(datasize DIV 2) operand2 = Vpart[m,part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result,e,esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMLS (by element)

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar, half-precision](#) , [Scalar, single-precision and double-precision](#) , [Vector, half-precision](#) and [Vector, single-precision and double-precision](#)

### Scalar, half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	0	L	M				Rm		0	1	0	1	H	0								Rd	
																	o2														

### Scalar, half-precision

```
FMLS <Hd>, <Hn>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UNDEFINED;
integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = () then UnallocatedEncoding();
integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

### Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	1	sz	L	M				Rm		0	1	0	1	H	0									Rd
																	o2														



Scalar, single-precision and double-precision

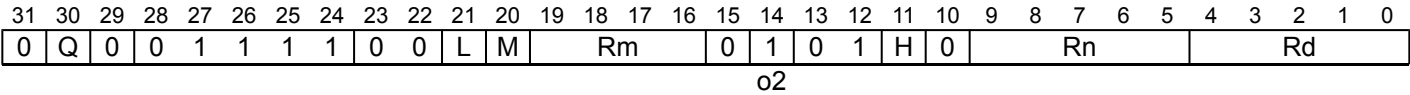
```
FMLS <V><d>, <V><n>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = when '11' UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Vector, half-precision  
(ARMv8.2)



Vector, half-precision

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

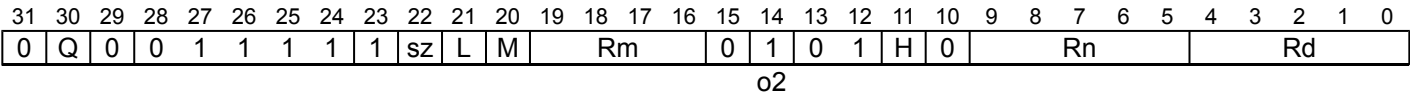
integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = () then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Vector, single-precision and double-precision



Vector, single-precision and double-precision

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
();

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.  
  
For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- <Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

- <index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FMLS (vector)

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm				0	0	0	0	1	1	Rn				Rd						
a																															

### Half-precision

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm				1	1	0	0	1	1	Rn				Rd						
op																															

### Single-precision and double-precision

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```

## Assembler Symbols

<Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMLSL, FMLSL2 (by element)

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

*ID\_AA64ISAR0\_EL1*.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: [FMLSL](#) and [FMLSL2](#)

### FMLSL

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	1	0	sz	L	M	Rm			0	1	0	0	H	0	Rn				Rd					
									sz	S																					

### FMLSL

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.[<index>]

```

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then ReservedValue('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
();
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q=='1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;

```

### FMLSL2

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	1	0	sz	L	M	Rm			1	1	0	0	H	0	Rn				Rd					
									sz	S																					

## FMLSL2

FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<H[<index>]>

```

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then ReservedValue('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
();
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q=='1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;

```

## Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
------	---

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
------	--

<b>&lt;Tb&gt;</b>	Is an arrangement specifier, encoded in “O”:
-------------------	--

Q	<Tb>
0	2H
1	4H

<b>&lt;Vm&gt;</b>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
-------------------	---

<code>&lt;index&gt;</code>	Is the element index, encoded in the "H:L:M" fields.
----------------------------	--

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n,part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FMLSL, FMLSL2 (vector)

Floating-point fused Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

*ID\_AA64ISAR0\_EL1.FHM* indicates whether this instruction is supported.

It has encodings from 2 classes: [FMLSL](#) and [FMLSL2](#)

### FMLSL

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0sz	1	Rm				1	1	1	0	1	1	Rn				Rd						
									S	sz																					

### FMLSL

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then ReservedValue(Rm);
if sz == '1' then UNDEFINED;
();
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;

```

### FMLSL2

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	0sz	1	Rm					1	1	0	0	1	1	Rn					Rd				
									S	sz																					



## FMLSL2

FMLS2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then ReservedValue(Rm);
if sz == '1' then UNDEFINED;
();
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;

```

## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
------	--

<b>&lt;Tb&gt;</b>	Is an arrangement specifier, encoded in “Q”:
-------------------	--

Q	<Tb>
0	2H
1	4H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n,part];
bits(datasize DIV 2) operand2 = Vpart[m,part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result,e,esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

(new)

## FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	1	1	d	e	f	g	h	Rd				

### Half-precision

FMOV <Vd>.<T>, #<imm>

```

if !HaveFP16Ext() then UNDEFINED;
integer rd =() then UnallocatedEncoding();
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;

imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,2):imm8<5:0>:Zeros(6);

imm = Replicate(imm16, datasize DIV 16);

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	0	1	d	e	f	g	h	Rd				

cmode

Single-precision (op == 0)

```
FMOV <Vd>.<T>, #<imm>
```

Double-precision (Q == 1 && op == 1)

```
FMOV <Vd>.2D, #<imm>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp MOVI;
  when '0xx01' operation = ImmediateOp MVNI;
  when '0xx10' operation = ImmediateOp ORR;
  when '0xx11' operation = ImmediateOp BIC;
  when '10x00' operation = ImmediateOp MOVI;
  when '10x01' operation = ImmediateOp MVNI;
  when '10x10' operation = ImmediateOp ORR;
  when '10x11' operation = ImmediateOp BIC;
  when '110x0' operation = ImmediateOp MOVI;
  when '110x1' operation = ImmediateOp MVNI;
  when '1110x' operation = ImmediateOp MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = if Q == '0' then UnallocatedEncoding();
    operation = ImmediateOp MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in A64 floating-point instructions*.

Operation

```
CheckFPAdvSIMDEnabled64();

V[rd] = imm;
```

## FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	0	0	0	0	0	1	0	0	0	0	Rn				Rd					
opc																															

### Half-precision (type == 11) (ARMv8.2)

```
FMOV <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FMOV <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FMOV <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
    when '11'
      if HaveFP16Ext() then
        datasize = 16;
      else
        UnallocatedEncoding() then
          datasize = 16;
        else
          UNDEFINED; ();

FPUnaryOp fpop;
case opc of
  when '00' fpop = FPUnaryOp_MOV;
  when '01' fpop = FPUnaryOp_ABS;
  when '10' fpop = FPUnaryOp_NEG;
  when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
  when FPUUnaryOp_MOV    result = operand;
  when FPUUnaryOp_ABS    result = FPAbs(operand);
  when FPUUnaryOp_NEG    result = FPNeg(operand);
  when FPUUnaryOp_SQRT   result = FPSqrt(operand, FPCR);

V[d] = result;
```

Internal version only: isa v30.25-v29.05, AdvSIMD v27.01-v26.0, pseudocode v85-xml-00bet8\_rc3-v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type	1	0	x	1	1	x	0	0	0	0	0	0	Rn				Rd						
rmode											opcode																				

**Half-precision to 32-bit (sf == 0 && type == 11 && rmode == 00 && opcode == 110)**  
(ARMv8.2)

FMOV <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && type == 11 && rmode == 00 && opcode == 110)**  
(ARMv8.2)

FMOV <Xd>, <Hn>

**32-bit to half-precision (sf == 0 && type == 11 && rmode == 00 && opcode == 111)**  
(ARMv8.2)

FMOV <Hd>, <Wn>

**32-bit to single-precision (sf == 0 && type == 00 && rmode == 00 && opcode == 111)**

FMOV <Sd>, <Wn>

**Single-precision to 32-bit (sf == 0 && type == 00 && rmode == 00 && opcode == 110)**

FMOV <Wd>, <Sn>

**64-bit to half-precision (sf == 1 && type == 11 && rmode == 00 && opcode == 111)**  
(ARMv8.2)

FMOV <Hd>, <Xn>

**64-bit to double-precision (sf == 1 && type == 01 && rmode == 00 && opcode == 111)**

FMOV <Dd>, <Xn>

**64-bit to top half of 128-bit (sf == 1 && type == 10 && rmode == 01 && opcode == 111)**

FMOV <Vd>.D[1], <Xn>

**Double-precision to 64-bit (sf == 1 && type == 01 && rmode == 00 && opcode == 110)**

FMOV <Xd>, <Dn>

**Top half of 128-bit to 64-bit (sf == 1 && type == 10 && rmode == 01 && opcode == 110)**

FMOV <Xd>, <Vn>.D[1]

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.



<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FMOV (scalar, immediate)

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD&FP destination register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	imm8								1	0	0	0	0	0	0	Rd						

### Half-precision (type == 11) (ARMv8.2)

FMOV <Hd>, #<imm>

### Single-precision (type == 00)

FMOV <Sd>, #<imm>

### Double-precision (type == 01)

FMOV <Dd>, #<imm>

```
integer d = UInt(Rd);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
    when '11'
      if HaveFP16Ext() then
        datasize = 16;
      else
        UnallocatedEncoding() then
          datasize = 16;
        else
          UNDEFINED;
();

bits(datasize) imm = VFPEExpandImm(imm8);
```

## Assembler Symbols

- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <imm>

Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in A64 floating-point instructions*.

## Operation

```
CheckFPAdvSIMDEnabled64();

V[d] = imm;
```

(old)

htmldiff from-

(new)

## FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	type	0	Rm				1	Ra				Rn				Rd								
									o1						o0																

### Half-precision (type == 11) (ARMv8.2)

FMSUB <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (type == 00)

FMSUB <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (type == 01)

FMSUB <Dd>, <Dn>, <Dm>, <Da>

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding() then
        datasize = 16;
    else
      UNDEFINED;
  ();

boolean opa_neg = (o1 == '1');
boolean opl_neg = (o0 != o1);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if opa_neg then operanda = FPNeg(operanda);
if opl_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FMUL (by element)

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar, half-precision](#) , [Scalar, single-precision and double-precision](#) , [Vector, half-precision](#) and [Vector, single-precision and double-precision](#)

### Scalar, half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	0	L	M	Rm				1	0	0	1	H	0	Rn				Rd					
U																															

### Scalar, half-precision

```
FMUL <Hd>, <Hn>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UNDEFINED;
integer idxsize = if H == '1' then 128 else 64;
integer index;
index = () then UnallocatedEncoding();
integer idxsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

### Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm				1	0	0	1	H	0	Rn				Rd					
U																															

Scalar, single-precision and double-precision

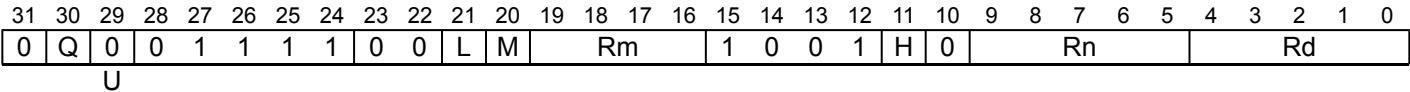
```
FMUL <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = when '11' UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Vector, half-precision
(ARMv8.2)



Vector, half-precision

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UNDEFINED;

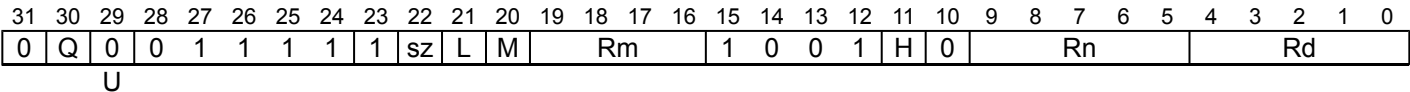
integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = () then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision



Vector, single-precision and double-precision

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
();

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.  
  
For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- <Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

- <index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.



For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FMUL (vector)

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm				0	0	0	1	1	1	Rn				Rd						

### Half-precision

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm				1	1	0	1	1	1	Rn				Rd						

### Single-precision and double-precision

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":



## FMUL (scalar)

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	1	1	1	1	0	type	1	Rm						0	0	0	0	1	0	Rn						Rd					
op																																	

### Half-precision (type == 11) (ARMv8.2)

```
FMUL <Hd>, <Hn>, <Hm>
```

### Single-precision (type == 00)

```
FMUL <Sd>, <Sn>, <Sm>
```

### Double-precision (type == 01)

```
FMUL <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding() then
        datasize = 16;
      else
        UNDEFINED;
  ();

boolean negated = (op == '1');
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);

if negated then result = FPNeg(result);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FMULX (by element)

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar, half-precision](#) , [Scalar, single-precision and double-precision](#) , [Vector, half-precision](#) and [Vector, single-precision and double-precision](#)

### Scalar, half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	0	L	M	Rm				1	0	0	1	H	0	Rn				Rd					
U																															

### Scalar, half-precision

```
FMULX <Hd>, <Hn>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UNDEFINED;
integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = () then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

### Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	sz	L	M	Rm				1	0	0	1	H	0	Rn				Rd					
U																															

Scalar, single-precision and double-precision

```
FMULX <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

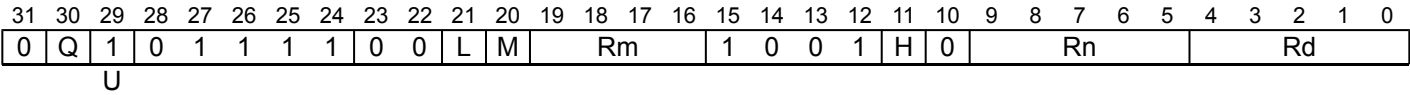
```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Vector, half-precision
(ARMv8.2)



Vector, half-precision

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UNDEFINED;

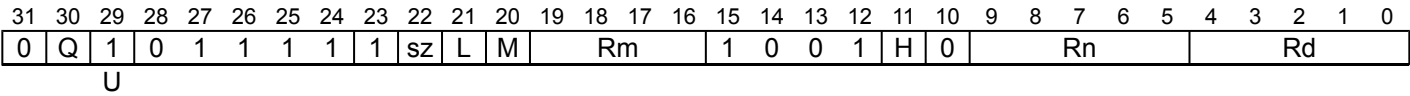
integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = () then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision



Vector, single-precision and double-precision

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
();

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.  
  
For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- <Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

- <index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.



For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FMULX

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	0				Rm		0	0	0	1	1	1				Rn				Rd		

### Scalar half precision

FMULX <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1				Rm		1	1	0	1	1	1				Rn				Rd		

### Scalar single-precision and double-precision

FMULX <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector half precision

(ARMv8.2)

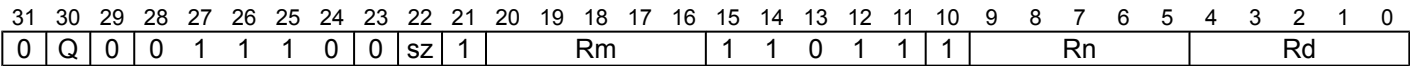
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0				Rm		0	0	0	1	1	1				Rn				Rd		

Vector half precision

FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



Vector single-precision and double-precision

FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v30.25-v29.05, AdvSIMD v27.01-v26.0, pseudocode v85-xml-00bet8\_rc3-v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FNEG (vector)

Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	1	1	0	Rn					Rd				
U																															

### Half-precision

FNEG <Vd>.<T>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	0	0	1	1	1	1	1	0	Rn				Rd				
U																															

### Single-precision and double-precision

FNEG <Vd>.<T>, <Vn>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

## Operation

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(new)

## FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	0	0	0	1	0	1	0	0	0	0	Rn					Rd				
opc																															

### Half-precision (type == 11) (ARMv8.2)

FNEG <Hd>, <Hn>

### Single-precision (type == 00)

FNEG <Sd>, <Sn>

### Double-precision (type == 01)

FNEG <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
    when '11'
      if HaveFP16Ext() then
        datasize = 16;
      else
        UnallocatedEncoding() then
          datasize = 16;
        else
          UNDEFINED; ();

FPUnaryOp fpop;
case opc of
  when '00' fpop = FPUnaryOp_MOV;
  when '01' fpop = FPUnaryOp_ABS;
  when '10' fpop = FPUnaryOp_NEG;
  when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
  when FPUUnaryOp_MOV    result = operand;
  when FPUUnaryOp_ABS    result = FPAbs(operand);
  when FPUUnaryOp_NEG    result = FPNeg(operand);
  when FPUUnaryOp_SQRT   result = FPSqrt(operand, FPCR);

V[d] = result;
```

Internal version only: isa v30.25-v29.05, AdvSIMD v27.01-v26.0, pseudocode v85-xml-00bet8\_rc3-v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## FNMADD

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
0	0	0	1	1	1	1	1	type	1	Rm				0	Ra				Rn				Rd																			
									o1																	o0																

### Half-precision (type == 11) (ARMv8.2)

```
FNMADD <Hd>, <Hn>, <Hm>, <Ha>
```

### Single-precision (type == 00)

```
FNMADD <Sd>, <Sn>, <Sm>, <Sa>
```

### Double-precision (type == 01)

```
FNMADD <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED;
        ();

boolean opa_neg = (o1 == '1');
boolean opl_neg = (o0 != o1);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if opa_neg then operanda = FPNeg(operanda);
if opl_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa **v30.25v29.05**, AdvSIMD **v27.01v26.0**, pseudocode **v85-xml-00bet8\_rc3v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FNMSUB

Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	type	1	Rm					1	Ra					Rn					Rd					
									o1		o0																				

### Half-precision (type == 11) (ARMv8.2)

FNMSUB <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (type == 00)

FNMSUB <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (type == 01)

FNMSUB <Dd>, <Dn>, <Dm>, <Da>

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED;
        ();

boolean opa_neg = (o1 == '1');
boolean opl_neg = (o0 != o1);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if opa_neg then operanda = FPNeg(operanda);
if opl_neg then operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;

```

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	Rm				1	0	0	0	1	0	Rn				Rd							
op																															

### Half-precision (type == 11) (ARMv8.2)

```
FNMUL <Hd>, <Hn>, <Hm>
```

### Single-precision (type == 00)

```
FNMUL <Sd>, <Sn>, <Sm>
```

### Double-precision (type == 01)

```
FNMUL <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UnallocatedEncoding() then
        datasize = 16;
      else
        UNDEFINED;
  ();

boolean negated = (op == '1');
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

- <Sd>Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn>Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm>Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);

if negated then result = FPNeg(result);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06:16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FRECPE

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#), [Scalar single-precision and double-precision](#), [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					

### Scalar half precision

FRECPE <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn				Rd					

### Scalar single-precision and double-precision

FRECPE <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					

Vector half precision

```
FRECPE <Vd>.<T>, <Vn>.<T>

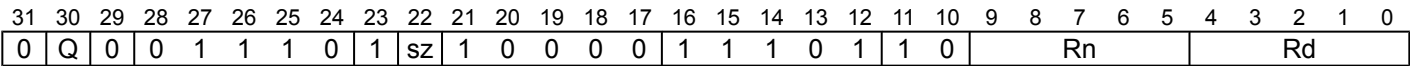
if !HaveFP16Ext() then UNDEFINED;

integer d =() then UnallocatedEncoding();

integer d = UInt (Rd);
integer n = UInt (Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FRECPE <Vd>.<T>, <Vn>.<T>

integer d = UInt (Rd);
integer n = UInt (Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt (sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.



Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FRECPS

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	0	Rm				0	0	1	1	1	1	Rn				Rd						

### Scalar half precision

FRECPS <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	Rm					1	1	1	1	1	1	Rn					Rd				

### Scalar single-precision and double-precision

FRECPS <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector half precision

(ARMv8.2)

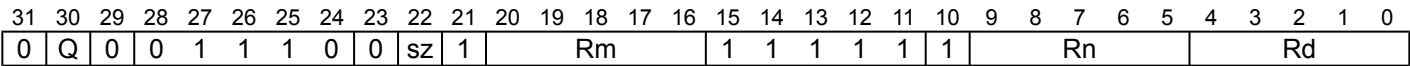
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm				0	0	1	1	1	1	Rn				Rd						

Vector half precision

```
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt (Rd);
integer n = UInt (Rn);
integer m = UInt (Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt (Rd);
integer n = UInt (Rn);
integer m = UInt (Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt (sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

FRECPX

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn				Rd					

Half-precision

FRECPX <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt (Rd) ;
integer n = UInt (Rn) ;

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn				Rd					

Single-precision and double-precision

FRECPX <V><d>, <V><n>

```
integer d = UInt (Rd) ;
integer n = UInt (Rn) ;

integer esize = 32 << UInt (sz) ;
integer datasize = esize;
integer elements = 1;
```

Assembler Symbols

- <Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn>

Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V>

Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D
- <d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPrecpX(element, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FRINT32X (vector)

Floating-point Round to 32-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Vector single-precision and double-precision (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	1	0	1	0	Rn				Rd					
U										op																					

### Vector single-precision and double-precision

```
FRINT32X <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);

V[d] = result;
```

**no old file**

**htmldiff from-**

**(new)**



## FRINT32X (scalar)

Floating-point Round to 32-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Floating-point (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	0	1	1	0	0	0	0	Rn						Rd					
type												op																					

#### Single-precision (type == 00)

```
FRINT32X <Sd>, <Sn>
```

#### Double-precision (type == 01)

```
FRINT32X <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '1x' UNDEFINED;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

### Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
  
bits(datasize) result;  
bits(datasize) operand = V[n];  
  
result = FPRoundIntN(operand, FPCR, rounding, intsize);  
  
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file

htmldiff from-

(new)

## FRINT32Z (vector)

Floating-point Round to 32-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Vector single-precision and double-precision (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	1	0	1	0	Rn				Rd					
U										op																					

### Vector single-precision and double-precision

```
FRINT32Z <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);

V[d] = result;
```

**no old file**

**htmldiff from-**

**(new)**

## FRINT32Z (scalar)

Floating-point Round to 32-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Floating-point (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	0	0	1	0	0	0	0	Rn						Rd					
type												op																					

#### Single-precision (type == 00)

```
FRINT32Z <Sd>, <Sn>
```

#### Double-precision (type == 01)

```
FRINT32Z <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '1x' UNDEFINED;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

### Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
  
bits(datasize) result;  
bits(datasize) operand = V[n];  
  
result = FPRoundIntN(operand, FPCR, rounding, intsize);  
  
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file

htmldiff from-

(new)

## FRINT64X (vector)

Floating-point Round to 64-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Vector single-precision and double-precision (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn				Rd					
U										op																					

### Vector single-precision and double-precision

FRINT64X <Vd>.<T>, <Vn>.<T>

```

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);

```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);

V[d] = result;

```

**no old file**

**htmldiff from-**

**(new)**



## FRINT64X (scalar)

Floating-point Round to 64-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Floating-point (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	1	1	1	0	0	0	0	Rn						Rd					
type												op																					

#### Single-precision (type == 00)

```
FRINT64X <Sd>, <Sn>
```

#### Double-precision (type == 01)

```
FRINT64X <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '1x' UNDEFINED;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

### Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
  
bits(datasize) result;  
bits(datasize) operand = V[n];  
  
result = FPRoundIntN(operand, FPCR, rounding, intsize);  
  
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file

htmldiff from-

(new)

## FRINT64Z (vector)

Floating-point Round to 64-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Vector single-precision and double-precision (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn				Rd					
U										op																					

### Vector single-precision and double-precision

```
FRINT64Z <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR, rounding, intsize);

V[d] = result;
```

**no old file**

**htmldiff from-**

**(new)**

## FRINT64Z (scalar)

Floating-point Round to 64-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Floating-point (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	1	0	1	0	0	0	0	Rn					Rd				
type											op																				

#### Single-precision (type == 00)

```
FRINT64Z <Sd>, <Sn>
```

#### Double-precision (type == 01)

```
FRINT64Z <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '1x' UNDEFINED;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR);
```

### Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
  
bits(datasize) result;  
bits(datasize) operand = V[n];  
  
result = FPRoundIntN(operand, FPCR, rounding, intsize);  
  
V[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file

htmldiff from-

(new)

## FRINTA (vector)

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn				Rd					
U								o2								o1															

### Half-precision

FRINTA <Vd>.<T>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  ();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn				Rd					
U								o2								o1															

Single-precision and double-precision

```
FRINTA <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```



## FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	1	0	0	1	0	0	0	0	Rn					Rd					
rmode																															

### Half-precision (type == 11) (ARMv8.2)

```
FRINTA <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTA <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTA <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UNDEFINED;
boolean exact = FALSE; else
UnallocatedEncoding();
boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding;
    when '101' UNDEFINED;
();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FRINTI (vector)

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn				Rd					
U				o2				o1																							

### Half-precision

```
FRINTI <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  ();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn				Rd					
U				o2				o1																							

## Single-precision and double-precision

```
FRINTI  <Vd>.<T>,  <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  ();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	$\langle T \rangle$
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
------	--

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:18.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

(new)

## FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	1	1	1	0	0	0	0	Rn					Rd						
rmode																															

### Half-precision (type == 11)

(ARMv8.2)

```
FRINTI <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTI <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTI <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UNDEFINED;
boolean exact = FALSE;
UnallocatedEncoding();
boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding;
    when '101' UNDEFINED;
    ();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:18Z2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn				Rd					
U								o2								o1															

### Half-precision

FRINTM <Vd>.<T>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  ();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn				Rd					
U								o2								o1															

Single-precision and double-precision

```
FRINTM <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```



## FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	0	1	0	0	0	0	0	Rn						Rd					
rmode																															

### Half-precision (type == 11) (ARMv8.2)

```
FRINTM <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTM <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTM <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UNDEFINED;
boolean exact = FALSE; else
UnallocatedEncoding();
boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding;
    when '101' UNDEFINED;
();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn				Rd					
U								o2							o1																

### Half-precision

FRINTN <Vd>.<T>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  ();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn				Rd					
U								o2							o1																

## Single-precision and double-precision

```
FRINTN  <Vd>.<T>,  <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  ();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	$\langle T \rangle$
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<b>&lt;Vn&gt;</b>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
-------------------	--

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:44Z2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	0	0	0	1	0	0	0	0											
rmode																								Rn				Rd			

### Half-precision (type == 11) (ARMv8.2)

```
FRINTN <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTN <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTN <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UNDEFINED;
boolean exact = FALSE;
UnallocatedEncoding();
boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding;
    when '101' UNDEFINED;
    ();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FRINTP (vector)

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn				Rd					
U				o2				o1																							

### Half-precision

FRINTP <Vd>.<T>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  ();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn				Rd					
U				o2				o1																							

Single-precision and double-precision

```
FRINTP <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```



## FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	0	0	1	1	0	0	0	0											
																Rn								Rd							
rmode																															

### Half-precision (type == 11) (ARMv8.2)

```
FRINTP <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTP <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTP <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UNDEFINED;
boolean exact = FALSE;
UnallocatedEncoding();
boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding;
    when '101' UNDEFINED;
    ();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:18+00:002018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FRINTX (vector)

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

When an [Inexact](#) exception is raised when the result value is not numerically equal to the corresponding input value, an [Inexact](#) exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn				Rd					
U								o2								o1															

### Half-precision

```
FRINTX <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  () ;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn				Rd					
U								o2								o1															

## Single-precision and double-precision

```
FRINTX <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  ();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
------	--

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:44Z2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

~~When an Inexact exception is raised when~~ the result value is not numerically equal to the input value, ~~an Inexact exception is raised.~~ ~~value.~~ A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	1	1	1	0	type	1	0	0	1	1	1	0	1	0	0	0	0	Rn						Rd					
rmode																																

### Half-precision (type == 11) (ARMv8.2)

```
FRINTX <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTX <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTX <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
    when '11'
      if HaveFP16Ext() then
        datasize = 16;
      else
        UNDEFINED;
  boolean exact = FALSE; else
  UnallocatedEncoding();
  boolean exact = FALSE;
  FPRounding rounding;
  case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding;
    when '101' UNDEFINED;
  ();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:18+00:002018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FRINTZ (vector)

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn				Rd					
U				o2				o1																							

### Half-precision

FRINTZ <Vd>.<T>, <Vn>.<T>

```

if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  ();
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);

```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn				Rd					
U				o2				o1																							

Single-precision and double-precision

```
FRINTZ <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UnallocatedEncoding;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```



## FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	1	0	1	1	1	0	0	0	0	Rn					Rd					
rmode																															

### Half-precision (type == 11) (ARMv8.2)

```
FRINTZ <Hd>, <Hn>
```

### Single-precision (type == 00)

```
FRINTZ <Sd>, <Sn>
```

### Double-precision (type == 01)

```
FRINTZ <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UNDEFINED;
boolean exact = FALSE; else
UnallocatedEncoding();
boolean exact = FALSE;
FPRounding rounding;
case rmode of
    when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding;
    when '101' UNDEFINED;
    ();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, exact);

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:18Z2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## FRSQRTE

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#), [Scalar single-precision and double-precision](#), [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					

### Scalar half precision

FRSQRTE <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn				Rd					

### Scalar single-precision and double-precision

FRSQRTE <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					

Vector half precision

FRSQRTE <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt (Rd);
integer n = UInt (Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn				Rd					

Vector single-precision and double-precision

FRSQRTE <Vd>.<T>, <Vn>.<T>

```
integer d = UInt (Rd);
integer n = UInt (Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt (sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FRSQRTS

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#), [Scalar single-precision and double-precision](#), [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	0	Rm				0	0	1	1	1	1	Rn				Rd						

### Scalar half precision

FRSQRTS <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	Rm				1	1	1	1	1	1	Rn				Rd						

### Scalar single-precision and double-precision

FRSQRTS <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector half precision

(ARMv8.2)

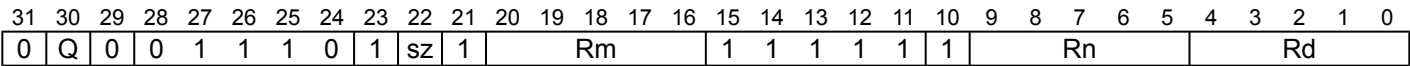
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm				0	0	1	1	1	1	Rn				Rd						

Vector half precision

```
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## FSQRT (vector)

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn				Rd					

### Half-precision

FSQRT <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn				Rd					

### Single-precision and double-precision

FSQRT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<I>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPSqrt(element, FPCR);

V[d] = result;

```

Internal version only: isa **v30.25-v29.05**, AdvSIMD **v27.01-v26.0**, pseudocode **v85-xml-00bet8\_rc3-v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	0	0	0	0	1	1	1	0	0	0	0	Rn						Rd				
opc																															

### Half-precision (type == 11) (ARMv8.2)

FSQRT <Hd>, <Hn>

### Single-precision (type == 00)

FSQRT <Sd>, <Sn>

### Double-precision (type == 01)

FSQRT <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED;()};

FPUnaryOp fpop;
case opc of
    when '00' fpop = FPUnaryOp_MOV;
    when '01' fpop = FPUnaryOp_ABS;
    when '10' fpop = FPUnaryOp_NEG;
    when '11' fpop = FPUnaryOp_SQRT;
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

- <Sd>Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn>Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

case fpop of
  when FPUUnaryOp_MOV    result = operand;
  when FPUUnaryOp_ABS    result = FPAbs(operand);
  when FPUUnaryOp_NEG    result = FPNeg(operand);
  when FPUUnaryOp_SQRT   result = FPSqrt(operand, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## FSUB (vector)

Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm				0	0	0	1	0	1	Rn				Rd						
U																															

### Half-precision

FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm					1	1	0	1	0	1	Rn					Rd				
U																															

### Single-precision and double-precision

FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) diff;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type	1	Rm				0	0	1	1	1	0	Rn				Rd							
																op															

### Half-precision (type == 11) (ARMv8.2)

FSUB <Hd>, <Hn>, <Hm>

### Single-precision (type == 00)

FSUB <Sd>, <Sn>, <Sm>

### Double-precision (type == 01)

FSUB <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                datasize = 16;
            else
                UnallocatedEncoding() then
                    datasize = 16;
            else
                UNDEFINED;
        ();
boolean sub_op = (op == '1');
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

- <Sd>Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn>Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm>Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

if sub_op then
    result = FPSub(operand1, operand2, FPCR);
else
    result = FPAdd(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)



## GMI

Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

### Integer (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	1	1	0	Xm				0	0	0	1	0	1	Xn				Xd						

### Integer

GMI <Xd>, <Xn|SP>, <Xm>

```
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

### Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
- <Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field.

### Operation

```
bits(64) address = if n == 31 then SP[] else X[n];
bits(64) mask = X[m];
bits(4) tag = AllocationTagFromAddress(address);

mask<UInt(tag)> = '1';
X[d] = mask;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## HINT

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	CRm			op2			1	1	1	1	1	0

## System

HINT #<imm>

```

SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111'
    SEE "XPACLR1";
  when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction();           // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction();   // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 100'
    op = SystemHintOp_CSDB;
  when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0'
    op = otherwise SystemHintOp_BTI;
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction();           // Instruction executes as NOP

```

## Assembler Symbols

- <imm>
- Is a 7-bit unsigned immediate, in the range 0 to 127 encoded in the "CRm:op2" field.
- The encodings that are allocated to architectural hint functionality are described in the "Hints" table in the "Index by Encoding".
- For allocated encodings of "CRm:op2":
- A disassembler will disassemble the allocated instruction, rather than the HINT instruction.
  - An assembler may support assembly of allocated encodings using HINT with the corresponding <imm> value, but it is not required to do so.

## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

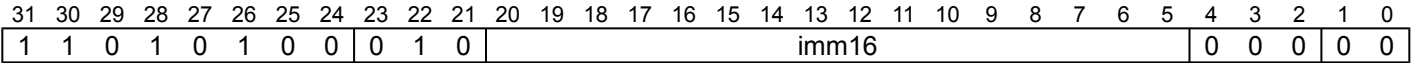
(old)

htmldiff from-

(new)

HLT

Halt ~~instruction.~~ ~~instruction~~ ~~A generates a Halt Instruction debug event.~~ HLT instruction can generate a Halt Instruction debug event, which causes entry into Debug state.



System

```
HLT #<imm>

if EDSCR.HDE == '0' || !HaltingAllowed() then UndefinedFault();
if() then UndefinedFault(); HaveBTIExt() then
    BTypeCompatible = TRUE;
```

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
Halt (DebugHalt_HaltInstruction);
```

HVC

Hypervisor Call causes an exception to EL2. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- At EL0, and Secure EL1.
- When `SCR_EL3.HCE` is set to 0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in `ESR_ELx`, using the EC value 0x16, and the value of the immediate argument.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	0	imm16																0	0	0	1	0

System

HVC #<imm>

bits(16) imm = imm16;

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && (!IsSecureEL2Enabled() && IsSecure())) then
    UNDEFINED;

hvc_enable = if(()) then UnallocatedEncoding();

hvc_enable = if(HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);
if hvc_enable == '0' then
    AArch64.UndefinedFault();
else
    AArch64.CallHypervisor(imm);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## INS (element)

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the [CPACR\\_ELI](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [MOV \(element\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	0	1	1	1	0	0	0	0	imm5					0	imm4					1	Rn					Rd				

### Advanced SIMD

INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer dst_index = if size > 3 then UnallocatedEncoding();
integer dst_index = UInt(imm5<4:size+1>);
integer src_index = UInt(imm4<3:size>);
integer idxdsize = if imm4<3> == '1' then 128 else 64;
// imm4<size-1:0> is IGNORED

integer esize = 8 << size;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in "imm5":

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index1> Is the destination element index encoded in "imm5":

imm5	<index1>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index2> Is the source element index encoded in "imm5:imm4":

imm5	<index2>
x0000	RESERVED
xxxx1	imm4<3:0>
xxx10	imm4<3:1>
xx100	imm4<3:2>
x1000	imm4<3>

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(128) result;

result = V[d];
Elem[result, dst_index, esize] = Elem[operand, src_index, esize];
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## INS (general)

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the [CPACR\\_ELI](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [MOV \(from general\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	0	imm5					0	0	0	1	1	1	Rn					Rd				

### Advanced SIMD

INS <Vd>.<Ts>[<index>], <R><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);

if size > 3 then UNDEFINED;
integer index = if size > 3 then UnallocatedEncoding();
integer index = UInt(imm5<4:size+1>);

integer esize = 8 << size;
integer datasize = 128;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in "imm5":

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in "imm5":

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<R> Is the width specifier for the general-purpose source register, encoded in "imm5":

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.



Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(datasize) result;

result = V[d];
Elem[result, index, esize] = element;
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## IRG

Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR\_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

### Integer (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	1	1	0	Xm				0	0	0	1	0	0	Xn				Xd						

### Integer

```
IRG <Xd|SP>, <Xn|SP>{, <Xm>}
```

```
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

### Assembler Symbols

- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
- <Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field. Defaults to XZR if absent.

### Operation

```
bits(64) operand = if n == 31 then SP[] else X[n];
bits(64) xm = X[m];
bits(16) exclude = xm<15:0> OR GCR_EL1.Exclude;

if AllocationTagAccessIsEnabled() then
    if GCR_EL1.RRND == '1' then
        RGSR_EL1 = bits(32) UNKNOWN;
        rtag = _ChooseRandomNonExcludedTag(exclude);
    else
        bits(4) start = RGSR_EL1.TAG;
        bits(4) offset = RandomTag();

        rtag = ChooseNonExcludedTag(start, offset, exclude);

        RGSR_EL1.TAG = rtag;
    else
        rtag = '0000';

bits(64) result = AddressWithAllocationTag(operand, rtag);

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see [Instruction Synchronization Barrier \(ISB\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm			1	1	0	1	1	1	1	1	
																											opc				

## System

ISB {<option>|#<imm>}

```

MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if otherwise HaveSBEExtUnallocatedEncoding() && CRm<3:0> == '0000' then
      op = ();
case CRm<3:2> of
  when '00' domain = MemBarrierOp_SB;
  else
    UNDEFINED;
case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;
  otherwise
    if CRm<3:2> == '01' then
      if CRm<3:2> == '00' then
        op = MemBarrierOp_PSSBB;
      elsif CRm<3:2> == '00' && opc == '00' then
        op = MemBarrierOp_SSBB;
      elsif elseif CRm<3:2> == '01' then
        op = HaveSBEExtMemBarrierOp_PSSBB() && CRm<3:2> == '00' && opc == '11' then
        op = MemBarrierOp_SB;
      else
        types = MBReqTypes_All;
        domain = MBReqDomain_FullSystem;

```

## Assembler Symbols

- <option> Specifies an optional limitation on the barrier operation. Values are:
- SY** Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.
- All other encodings of CRm are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.
- <imm> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

```
case op of
  when MemBarrierOp_DSBDDataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier();
  when MemBarrierOp_SSBBSpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA();
  when (); MemBarrierOp_SBSpeculationBarrier();
```

Internal version only: isa v30.25-v29.05, AdvSIMD v27.01-v26.0, pseudocode v85-xml-00bet8\_rc3-v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	x	x	1	x	size		Rn				Rt					
L										opcode																					

### One register (opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>]
```

### Two registers (opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

### Three registers (opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

### Four registers (opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm				x	x	1	x	size			Rn				Rt					
L											opcode																				

One register, immediate offset (Rm == 11111 && opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED; otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
            endif
        endfor
    endfor

if wback then
    if m != 31 then
        offs = X[m];
    endif
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
    endif
endif
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	x	x	0	S	size	Rn					Rt					
L										R	opcode																				

### 8-bit (opcode == 000)

```
LD1 { <Vt>.B } [<index>], [<Xn|SP>]
```

### 16-bit (opcode == 010 && size == x0)

```
LD1 { <Vt>.H } [<index>], [<Xn|SP>]
```

### 32-bit (opcode == 100 && size == 00)

```
LD1 { <Vt>.S } [<index>], [<Xn|SP>]
```

### 64-bit (opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D } [<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					x	x	0	S	size	Rn					Rt					
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<I>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
    ({});
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

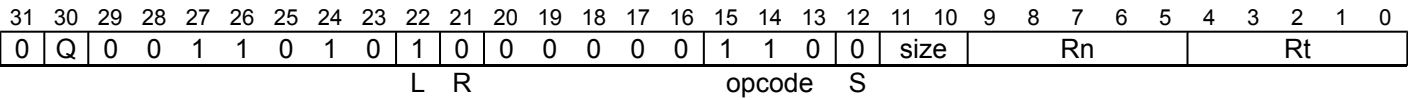
LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

No offset

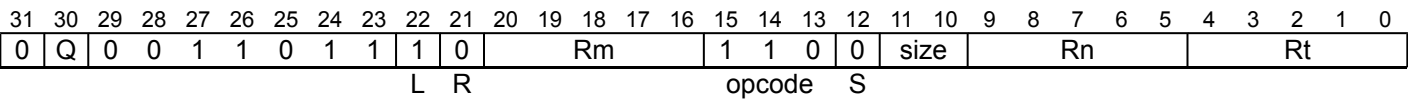


No offset

```
LD1R { <Vt>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index



Immediate offset (Rm == 11111)

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

- <Vt>Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T>Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#1
01	#2
10	#4
11	#8

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
    ();
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LD2 (multiple structures)

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see [LD3 \(multiple structures\)](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	size	Rn				Rt						
L										opcode																					

### No offset

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm				1	0	0	0	size	Rn				Rt							
L										opcode																					

### Immediate offset (Rm == 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D



- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "Q":
 

Q	<imm>
0	#16
1	#32
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

### Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;      // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;      // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;      // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;      // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;      // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;      // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;      // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED; otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
            endif
        endfor
    endfor

if wback then
    if m != 31 then
        offs = X[m];
    endif
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
    endif
endif
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LD2 (single structure)

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	0	S	size	Rn			Rt							
L										R	opcode																				

### 8-bit (opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

### 16-bit (opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

### 32-bit (opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

### 64-bit (opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm					x	x	0	S	size	Rn					Rt					
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
    ();
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LD2R

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	size	Rn					Rt					
L R										opcode S																					

### No offset

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm					1	1	0	0	size	Rn					Rt					
L										R	opcode										S										

### Immediate offset (Rm == 11111)

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "size":

size	<imm>
00	#2
01	#4
10	#8
11	#16

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
      if size<0> == '0' then
        index = UInt(Q:S); // S[0-3]
      else
        if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
      else
        if S == '1' then UNDEFINED;
    end if
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```



## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

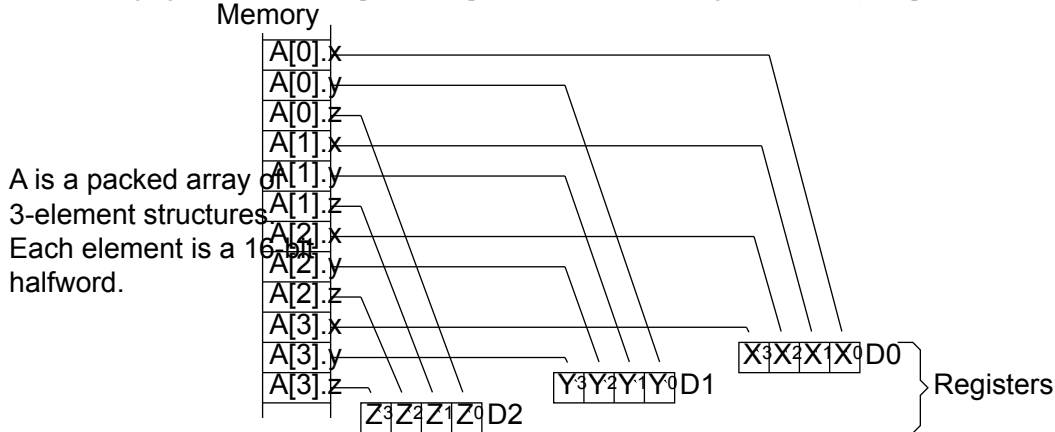
htmldiff from-

(new)

## LD3 (multiple structures)

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:.



Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	size	Rn				Rt						
L										opcode																					

### No offset

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm				0	1	0	0	size	Rn				Rt							
L											opcode																				

### Immediate offset (Rm == 11111)

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#24
1	#48

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED; otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
            endif
        endfor
    endfor

if wback then
    if m != 31 then
        offs = X[m];
    endif
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
    endif
endif
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LD3 (single structure)

Load single 3-element structure to one lane of three registers). This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	x	x	1	S	size	Rn			Rt							
L R										opcode																					

### 8-bit (opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

### 16-bit (opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

### 32-bit (opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

### 64-bit (opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					x	x	1	S	size	Rn					Rt					
L R											opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>], #3
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>], #6
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>], #12
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>], #24
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
    ();
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	1	1	1	0	size	Rn						Rt					
L										R	opcode										S											

### No offset

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					1	1	1	0	size	Rn					Rt					
L										R	opcode										S										

### Immediate offset (Rm == 11111)

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "size":

size	<imm>
00	#3
01	#6
10	#12
11	#24

- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```

integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
    ();
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LD4 (multiple structures)

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	size	Rn				Rt						
L										opcode																					

### No offset

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm				0	0	0	0	size	Rn				Rt							
L										opcode																					

### Immediate offset (Rm == 11111)

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

### Register offset (Rm != 11111)

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

### Assembler Symbols

- <Vt>Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T>Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "Q":
- | Q | <imm> |
|---|-------|
| 0 | #32   |
| 1 | #64   |
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED; otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
            endif
        endfor
    endfor

if wback then
    if m != 31 then
        offs = X[m];
    endif
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
    endif
endif
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	1	S	size	Rn				Rt						
L										R	opcode																				

### 8-bit (opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

### 16-bit (opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

### 32-bit (opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

### 64-bit (opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm				x	x	1	S	size	Rn				Rt							
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.



## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);          // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>);      // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S);            // S[0-3]
    else
      if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
    ();
    index = UInt(Q);                // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "size":

size	<imm>
00	#4
01	#8
10	#16
11	#32

- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```

integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
      if size<0> == '0' then
        index = UInt(Q:S); // S[0-3]
      else
        if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
      else
        if S == '1' then UNDEFINED;
    end if
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

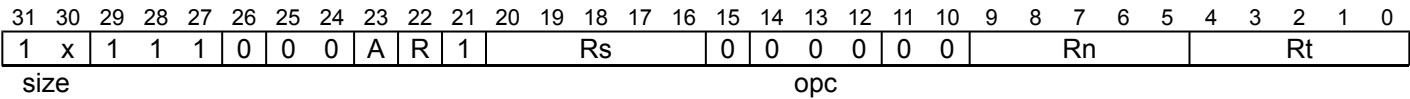
- If the destination register is not one of WZR or XZR, LDADDA and LDADDAL load from memory with acquire semantics.
- LDADDL and LDADDAL store to memory with release semantics.
- LDADD has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STADD, STADDL](#).

Integer  
(ARMv8.1)



### 32-bit LDADD (size == 10 && A == 0 && R == 0)

```
LDADD <Ws>, <Wt>, [<Xn|SP>]
```

### 32-bit LDADDA (size == 10 && A == 1 && R == 0)

```
LDADDA <Ws>, <Wt>, [<Xn|SP>]
```

### 32-bit LDADDAL (size == 10 && A == 1 && R == 1)

```
LDADDAL <Ws>, <Wt>, [<Xn|SP>]
```

### 32-bit LDADDL (size == 10 && A == 0 && R == 1)

```
LDADDL <Ws>, <Wt>, [<Xn|SP>]
```

### 64-bit LDADD (size == 11 && A == 0 && R == 0)

```
LDADD <Xs>, <Xt>, [<Xn|SP>]
```

### 64-bit LDADDA (size == 11 && A == 1 && R == 0)

```
LDADDA <Xs>, <Xt>, [<Xn|SP>]
```

### 64-bit LDADDAL (size == 11 && A == 1 && R == 1)

```
LDADDAL <Xs>, <Xt>, [<Xn|SP>]
```

### 64-bit LDADDL (size == 11 && A == 0 && R == 1)

```
LDADDL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

- <Xt>Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STADD, STADDL</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDLB and LDADDALB store to memory with release semantics.
- LDADDB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STADDB, STADDLB](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs				0	0	0	0	0	0	Rn				Rt						
size											opc																				

#### LDADDAB (A == 1 && R == 0)

```
LDADDAB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDADDALB (A == 1 && R == 1)

```
LDADDALB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDADDB (A == 0 && R == 0)

```
LDADDB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDADDLB (A == 0 && R == 1)

```
LDADDLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = () then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

- <Ws>
- Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt>
- Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>
- Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STADDB, STADDLB</a>	<code>A == '0' &amp;&amp; Rt == '11111'</code>

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAH and LDADDALH load from memory with acquire semantics.
- LDADDLH and LDADDALH store to memory with release semantics.
- LDADDH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STADDH, STADDLH](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs				0	0	0	0	0	0	Rn				Rt						
size											opc																				

#### LDADDAH (A == 1 && R == 0)

```
LDADDAH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDADDALH (A == 1 && R == 1)

```
LDADDALH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDADDH (A == 0 && R == 0)

```
LDADDH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDADDLH (A == 0 && R == 1)

```
LDADDLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = () then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STADDH, STADDLH</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8](#) ~~rc3~~[v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

# LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

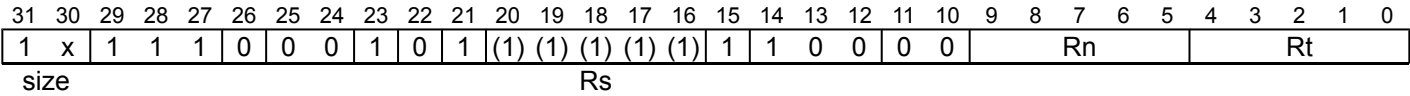
The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

## Integer (ARMv8.3)



### 32-bit (size == 10)

```
LDAPR <Wt>, [<Xn|SP> {,#0}]
```

### 64-bit (size == 11)

```
LDAPR <Xt>, [<Xn|SP> {,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

AccType acctype = AccType_ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25v29.05~~, AdvSIMD ~~v27.01v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3v35.3~~; Build timestamp: ~~2018-09-13T13:20:18-06:16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDAPRB

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

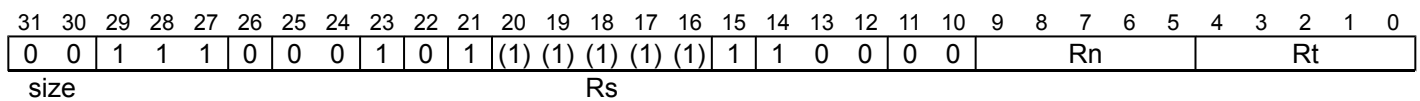
The instruction has memory ordering semantics as described in [Load-Acquire, Load-AcquirePC, and Store-Release](#), except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see [Load/Store addressing modes](#).

### Integer (ARMv8.3)



### Integer

```
LDAPRB <Wt>, [<Xn|SP> {, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release
```

```
AccType acctype = AccType.ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

(old)

htmldiff from-

(new)



## LDAPRH

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

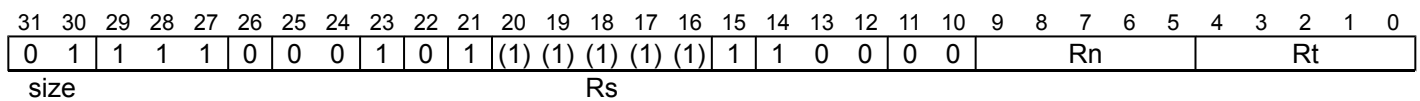
The instruction has memory ordering semantics as described in [Load-Acquire, Load-AcquirePC, and Store-Release](#), except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see [Load/Store addressing modes](#).

### Integer (ARMv8.3)



### Integer

```
LDAPRH <Wt>, [<Xn|SP> {, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release
```

```
AccType acctype = AccType.ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

(old)

htmldiff from-

(new)

## LDAPUR

Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in [Load-Acquire, Load-AcquirePC, and Store-Release](#), except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	0	1	1	0	0	1	0	1	0	imm9									0	0	Rn				Rt									
size								opc																											

### 32-bit (size == 10)

```
LDAPUR <Wt>, [<Xn|SP>{, #<simm>}]
```

### 64-bit (size == 11)

```
LDAPUR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

LDAPURB

Load-Acquire RCpc Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	1	0	0	1	0	1	0	imm9									0	0	Rn				Rt						
size								opc																								

Unscaled offset

LDAPURB <Wt>, [<Xn|SP>{, #<simm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.



## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:20:09~~2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# LDAPURH

Load-Acquire RCpc Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	1	0	1	0	imm9									0	0	Rn				Rt					
size								opc																							

## Unscaled offset

LDAPURH <Wt>, [<Xn|SP>{, #<sim>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```



```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# LDAPURSB

Load-Acquire RCpc Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	1	x	0	imm9									0	0	Rn				Rt					
size										opc																					

## 32-bit (opc == 11)

```
LDAPURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

## 64-bit (opc == 10)

```
LDAPURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# LDAPURSH

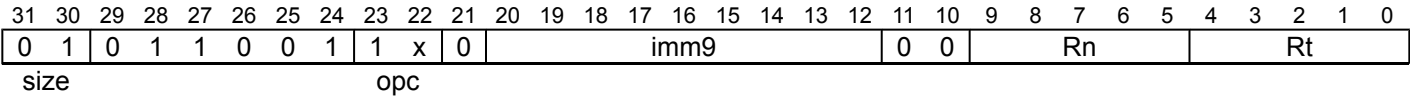
Load-Acquire RCpc Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.



## 32-bit (opc == 11)

```
LDAPURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

## 64-bit (opc == 10)

```
LDAPURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```





```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE;          // writeback is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE;           // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE;          // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding UNDEFINED;
    ();
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# LDAPURSW

Load-Acquire RCpc Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	1	1	0	0	imm9						0	0	Rn				Rt								
size								opc																							

## Unscaled offset

```
LDAPURSW <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size				L						Rs				o0		Rt2															

### 32-bit (size == 10)

```
LDAR <Wt>, [<Xn|SP>{, #0}]
```

### 64-bit (size == 11)

```
LDAR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25~~v29.05~~, AdvSIMD v27.01~~v26.0~~, pseudocode v85-xml-00bet8\_rc3~~v35.3~~; Build timestamp: 2018-09-13T13~~2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt					
size								L		Rs				o0		Rt2																

### No offset

```
LDARB <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release
```

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

### Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

(old)

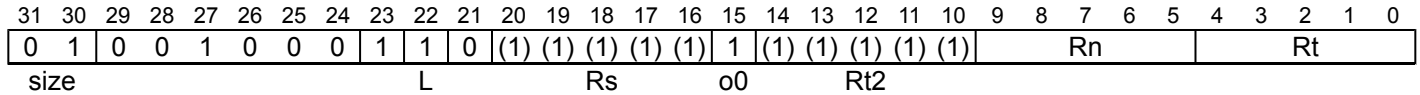
htmldiff from-

(new)

## LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.



### No offset

```
LDARH <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release
```

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

### Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)	1	Rt2				Rn				Rt						
								L			Rs					o0															

### 32-bit (sz == 0)

```
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

### 64-bit (sz == 1)

```
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release
```

```
AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDAXP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.





```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF        UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP          EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF        UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP          EndOfInstruction();
        if s == n && n != 31 then
            Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
            assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
            case c of
                when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
                when Constraint_NONE        rn_unknown = FALSE;   // address is original base
                when Constraint_UNDEF        UnallocatedEncoding    UNDEFINED;
                when Constraint_NOP          EndOfInstruction();

            );

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L			Rs				o0		Rt2														

### 32-bit (size == 10)

```
LDAXR <Wt>, [<Xn|SP>{, #0}]
```

### 64-bit (size == 11)

```
LDAXR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
        if s == n && n != 31 then
            Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
            assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
            case c of
                when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
                when Constraint_NONE        rn_unknown = FALSE;   // address is original base
                when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
                when Constraint_NOP        EndOfInstruction();

            );

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size								L			Rs				o0		Rt2														

### No offset

```
LDAXRB <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType ORDEREDATOMIC else AccType ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.





```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

    when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size								L			Rs				o0		Rt2														

### No offset

```
LDAXRH <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType ORDEREDATOMIC else AccType ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

    when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

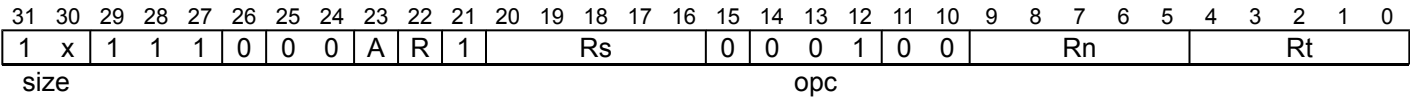
- If the destination register is not one of WZR or XZR, LDCLRA and LDCLRAL load from memory with acquire semantics.
- LDCLRL and LDCLRAL store to memory with release semantics.
- LDCLR has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STCLR, STCLRL](#).

Integer  
(ARMv8.1)



### 32-bit LDCLR (size == 10 && A == 0 && R == 0)

LDCLR <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDCLRA (size == 10 && A == 1 && R == 0)

LDCLRA <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDCLRAL (size == 10 && A == 1 && R == 1)

LDCLRAL <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDCLRL (size == 10 && A == 0 && R == 1)

LDCLRL <Ws>, <Wt>, [<Xn|SP>]

### 64-bit LDCLR (size == 11 && A == 0 && R == 0)

LDCLR <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDCLRA (size == 11 && A == 1 && R == 0)

LDCLRA <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDCLRAL (size == 11 && A == 1 && R == 1)

LDCLRAL <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDCLRL (size == 11 && A == 0 && R == 1)

LDCLRL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.



- <Xt>Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STCLR, STCLRL</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRALB load from memory with acquire semantics.
- LDCLRLB and LDCLRALB store to memory with release semantics.
- LDCLRB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STCLRB, STCLRLB](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs				0	0	0	1	0	0	Rn				Rt						
size											opc																				

#### LDCLRAB (A == 1 && R == 0)

```
LDCLRAB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDCLRALB (A == 1 && R == 1)

```
LDCLRALB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDCLRB (A == 0 && R == 0)

```
LDCLRB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDCLRLB (A == 0 && R == 1)

```
LDCLRLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = () then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STCLRB</a> , <a href="#">STCLRLB</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)bet8\_rc3~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old) [htmldiff](#) from- (new)

## LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAH and LDCLRALH load from memory with acquire semantics.
- LDCLRLH and LDCLRALH store to memory with release semantics.
- LDCLRH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STCLRH, STCLRLH](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs				0	0	0	1	0	0	Rn				Rt						
size											opc																				

#### LDCLRAH (A == 1 && R == 0)

```
LDCLRAH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDCLRALH (A == 1 && R == 1)

```
LDCLRALH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDCLRH (A == 0 && R == 0)

```
LDCLRH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDCLRLH (A == 0 && R == 1)

```
LDCLRLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = () then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STCLRH</a> , <a href="#">STCLRLH</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)bet8\_rc3~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

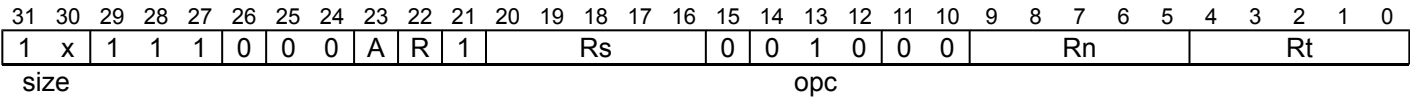
- If the destination register is not one of WZR or XZR, LDEORA and LDEORAL load from memory with acquire semantics.
- LDEORL and LDEORAL store to memory with release semantics.
- LDEOR has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STEOR, STEORL](#).

Integer  
(ARMv8.1)



### 32-bit LDEOR (size == 10 && A == 0 && R == 0)

LDEOR <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDEORA (size == 10 && A == 1 && R == 0)

LDEORA <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDEORAL (size == 10 && A == 1 && R == 1)

LDEORAL <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDEORL (size == 10 && A == 0 && R == 1)

LDEORL <Ws>, <Wt>, [<Xn|SP>]

### 64-bit LDEOR (size == 11 && A == 0 && R == 0)

LDEOR <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDEORA (size == 11 && A == 1 && R == 0)

LDEORA <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDEORAL (size == 11 && A == 1 && R == 1)

LDEORAL <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDEORL (size == 11 && A == 0 && R == 1)

LDEORL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

- <Xt>Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STEOR</a> , <a href="#">STEORL</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STEORB, STEORLB](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs				0	0	1	0	0	0	Rn				Rt						
size											opc																				

#### LDEORAB (A == 1 && R == 0)

```
LDEORAB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDEORALB (A == 1 && R == 1)

```
LDEORALB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDEORB (A == 0 && R == 0)

```
LDEORB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDEORLB (A == 0 && R == 1)

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = () then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STEORB</a> , <a href="#">STEORLB</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)bet8\_rc3~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old) [htmldiff](#) from- (new)

## LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAH and LDEORALH load from memory with acquire semantics.
- LDEORLH and LDEORALH store to memory with release semantics.
- LDEORH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STEORH, STEORLH](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs				0	0	1	0	0	0	Rn				Rt						
size											opc																				

#### LDEORAH (A == 1 && R == 0)

```
LDEORAH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDEORALH (A == 1 && R == 1)

```
LDEORALH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDEORH (A == 0 && R == 0)

```
LDEORH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDEORLH (A == 0 && R == 1)

```
LDEORLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = () then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STEORH</a> , <a href="#">STEORLH</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8](#) ~~rc3~~[v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old) [htmldiff](#) from- (new)

## LDG

Load Allocation Tag loads an Allocation Tag from a memory address, generates an address with the Logical Address Tag generated from the loaded Allocation Tag, and writes the result to the destination register. The address used for the load is calculated from the source register and an immediate signed offset scaled by the Tag granule.

### Integer (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	1	1	imm9									0	0	Xn				Xt					

### Integer

```
LDG <Xt>, [<Xn|SP>{, #<sim>}]
```

```
integer t = UInt(Xt);
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
```

### Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<sim>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

### Operation

```
bits(64) address;
bits(4) tag;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
address = Align(address, TAG_GRANULE);

tag = MemTag[address];
address = AddressWithAllocationTag(address, tag);

X[t] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDGV

Load Tag Vector loads an IMPLEMENTATION DEFINED number of Allocation Tags from the naturally aligned array of 16 Allocation Tags which includes a tag whose address is the address in the source register, and writes them to the destination register. Bits of the destination register which do not store a tag are set to 0. The Allocation Tag at the address in the source register is always loaded, and the first source register is updated to the address of the first Allocation Tag at an address higher than the original address that was not loaded.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

### Integer (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	Xn						Xt					

### Integer

```
LDGV <Xt>, [<Xn|SP>]!
```

```
integer t = UInt(Xt);
integer n = UInt(Xn);
boolean wback = TRUE;
boolean wb_unknown = FALSE;
```

### Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

## Operation

```
bits(64) data = Zeros(64);
bits(64) address;
integer count;

if n == t then
  c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPLD);
  assert c IN {Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};

  case c of
    when Constraint\_WBSUPPRESS  wback = FALSE; // writeback is suppressed
    when Constraint\_UNKNOWN    wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint\_UNDEF      UnallocatedEncoding();
    when Constraint\_NOP        EndOfInstruction();

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

(address, count) = ImpDefTagArrayStartAndCount(address);

for i = 0 to count-1
  integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);
  bits(4) tag = MemTag[address];
  data<(index*4)+3:(index*4)> = tag;
  address = address + TAG_GRANULE;

X[t] = data;

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file

htmldiff from-

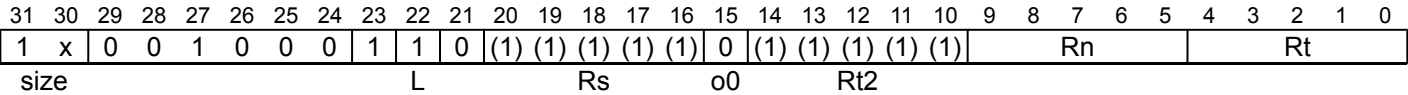
(new)

# LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in [Load LOAcquire, Store LORelease](#). For information about memory accesses, see [Load/Store addressing modes](#).

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

## No offset (ARMv8.1)



## 32-bit (size == 10)

```
LDLAR <Wt>, [<Xn|SP>{, #0}]
```

## 64-bit (size == 11)

```
LDLAR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:20:04Z~~2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in [Load LOAcquire, Store LORelease](#). For information about memory accesses, see [Load/Store addressing modes](#).

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

### No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn				Rt					
size				L				Rs				o0				Rt2															

### No offset

```
LDLARB <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release
```

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDLARH

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in [Load LOAcquire, Store LORelease](#). For information about memory accesses, see [Load/Store addressing modes](#).

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

### No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L		Rs				o0		Rt2															

### No offset

```
LDLARH <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25~~v29.05~~, AdvSIMD v27.01~~v26.0~~, pseudocode v85-xml-00bet8\_rc3~~v35.3~~; Build timestamp: 2018-09-13T13~~2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

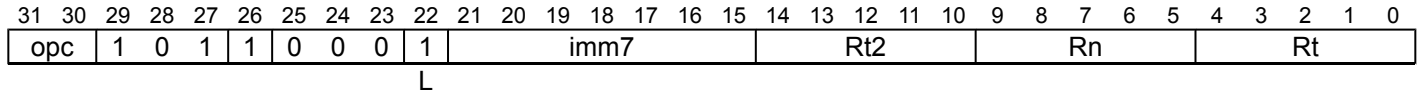
(old)	htmldiff from-	(new)
-------	----------------	-------

## LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see [Load/Store SIMD and Floating-point Non-temporal pair](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### 32-bit (opc == 00)

```
LDNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDNP \(SIMD&FP\)](#).

## Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.  For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VECSTREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 +if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == && t == t2 then MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding    UNDEFINED;
    ();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0, dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0, dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t] = data1;
        V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see [Load/Store addressing modes](#). For information about Non-temporal pair instructions, see [Load/Store Non-temporal pair](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
x		0		1		0		1		0		0		0		0		1		imm7							Rt2				Rn				Rt			
opc										L																												

### 32-bit (opc == 00)

```
LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 10)

```
LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDNP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + if opc<0> == '1' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == && t == t2 then MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;        // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding    UNDEFINED;
    ();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0      , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0      , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        X[t]  = data1;
        X[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	0	1	1	imm7							Rt2					Rn					Rt				
L																															

#### 32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>], #<imm>
```

#### 64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

#### 128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1 0 1		1 0 1 1		1		imm7								Rt2					Rn					Rt					
L																															

#### 32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

#### 64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

#### 128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	1	0	1	imm7							Rt2					Rn					Rt				

### 32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDP \(SIMD&FP\)](#).

## Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.  For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.  For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.  For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.  For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == t1 && t == t2 then MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;        // result is UNKNOWN
        when Constraint_UNDEF       UnallocatedEncoding      UNDEFINED;
    when Constraint_NOP            EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0, dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0, dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t] = data1;
        V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	0	1	0	1	0	0	0	1	1	imm7							Rt2				Rn				Rt						
opc										L																					

#### 32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

#### 64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
x		0	1		0	1		0	0		1	1		1	imm7							Rt2				Rn				Rt			
opc										L																							

#### 32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

#### 64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x		0	1		0	1		0	0		1	imm7					Rt2					Rn					Rt				
opc										L																					

#### 32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

#### 64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	<p>For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/8.</p>

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```





```

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

boolean wb_unknown = FALSE;

if memop == && wback && (t == n || t2 == n) && n != 31 then MemOp_LOAD && wback && (t == n || t2 == n) &
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0, dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0, dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;

```

```

    if signed then
        X[t]  = SignExtend(data1, 64);
        X[t2] = SignExtend(data2, 64);
    else
        X[t]  = data1;
        X[t2] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	0	1	1	imm7							Rt2				Rn				Rt						
opc										L																					

### Post-index

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	1	1	imm7							Rt2				Rn				Rt						
opc										L																					

### Pre-index

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	0	1	imm7							Rt2				Rn				Rt						
opc										L																					

### Signed offset

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDPSW](#).

## Assembler Symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```



```

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

boolean wb_unknown = FALSE;

if memop == && wback && (t == n || t2 == n) && n != 31 then MemOp_LOAD && wback && (t == n || t2 == n) &
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0, dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0, dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;

```

```

    if signed then
        X[t]  = SignExtend(data1, 64);
        X[t2] = SignExtend(data2, 64);
    else
        X[t]  = data1;
        X[t2] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25~~v29.05~~, AdvSIMD v27.01~~v26.0~~, pseudocode v85-xml-00bet8\_rc3~~v35.3~~; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDR (immediate, SIMD&FP)

Load SIMD&FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	1	0	imm9										0	1	Rn					Rt			
opc																															

#### 8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>], #<sim>
```

#### 16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>], #<sim>
```

#### 32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>], #<sim>
```

#### 64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>], #<sim>
```

#### 128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	1	0	imm9										1	1	Rn					Rt			
opc																															



8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>, #<sim>]!
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, #<sim>]!
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, #<sim>]!
```

64-bit (size == 11 && opc == 01)

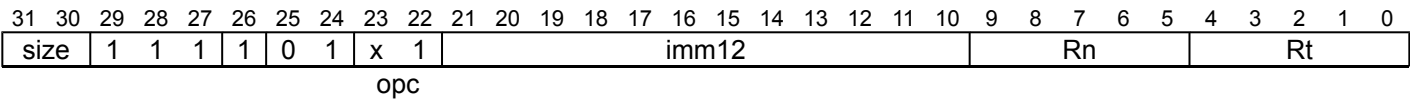
```
LDR <Dt>, [<Xn|SP>, #<sim>]!
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>.size);
if scale > 4 then UNDEFINED;
bits(64) offset = if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>.size);
if scale > 4 then UNDEFINED;
bits(64) offset = if scale > 4 then UnallocatedEncoding();
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	<p>For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.</p> <p>For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/2.</p> <p>For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/4.</p> <p>For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/8.</p> <p>For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/16.</p>

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation

```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDR (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#). The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	1	0	imm9									0	1	Rn				Rt									
size										opc																									

#### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>], #<sim>
```

#### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	1	0	imm9									1	1	Rn				Rt									
size										opc																									

#### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>, #<sim>]!
```

#### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
1		x		1		1		1		0		0		1		0		1		imm12												Rn					Rt				
size										opc																															

### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDR \(immediate\)](#).

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.  For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## LDR (literal, SIMD&FP)

Load SIMD&FP Register (PC-relative literal). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		0	1	1	1	0	0	imm19																	Rt						

### 32-bit (opc == 00)

```
LDR <St>, <label>
```

### 64-bit (opc == 01)

```
LDR <Dt>, <label>
```

### 128-bit (opc == 10)

```
LDR <Qt>, <label>
```

```
integer t = UInt(Rt);
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 16;
    when '11'
        UNDEFINED;
offset = when '11' UnallocatedEncoding();
offset = SignExtend(imm19:'00', 64);
```

## Assembler Symbols

<Dt>	Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<label>	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

## Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

CheckFPAdvSIMDEnabled64();

data = Mem[address, size, AccType VEC];
V[t] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDR (register, SIMD&FP)

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
size		1		1		1		1		0		0		x		1		1		Rm				option		S		1		0		Rn				Rt		
opc																																						

### 8-fsreg,LDR-8-fsreg (size == 00 && opc == 01 && option != 011)

```
LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### 8-fsreg,LDR-8-fsreg (size == 00 && opc == 01 && option == 011)

```
LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### 16-fsreg,LDR-16-fsreg (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 32-fsreg,LDR-32-fsreg (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-fsreg,LDR-64-fsreg (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 128-fsreg,LDR-128-fsreg (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED; // sub-word indexif scale > 4 then
UnallocatedEncoding();
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#4

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if(m, extend_type, shift); HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

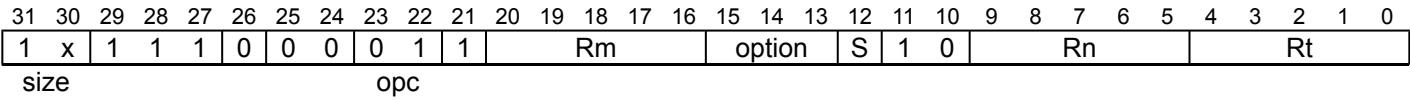
(old)

htmldiff from-

(new)

LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see [Load/Store addressing modes](#).



32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word indexif option<1> == '0' then
UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm>

When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm>

When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```





```

bits(64) offset = ExtendReg(m, extend_type, shift);
if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding  UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDRAA, LDRAB

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for LDRAA, and key B is used for LDRAB.

If the authentication passes, the PE behaves the same as for an LDR instruction. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see [Load/Store addressing modes](#).

### Unscaled offset

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	M	S	1	imm9									W	1	Rn				Rt					
size																															

#### Key A, offset (M == 0 && W == 0)

```
LDRAA <Xt>, [<Xn|SP>{, #<sim>}]
```

#### Key A, pre-indexed (M == 0 && W == 1)

```
LDRAA <Xt>, [<Xn|SP>{, #<sim>}]!
```

#### Key B, offset (M == 1 && W == 0)

```
LDRAB <Xt>, [<Xn|SP>{, #<sim>}]
```

#### Key B, pre-indexed (M == 1 && W == 1)

```
LDRAB <Xt>, [<Xn|SP>{, #<sim>}]!
```

```
if !HavePACExt() || size != '11' then UNDEFINED;
integer t =() || size != '11' then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) S10 = S:imm9;
integer scale = 3;
bits(64) offset = LSL(SignExtend(S10, 64), scale);
```

### Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <sim>/8.

## Operation

```
bits(64) address;
bits(64) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPLD);
    assert c IN {Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint\_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint\_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if use_key_a then
    address = AuthDA(address, X[31]);
else
    address = AuthDB(address, X[31]);

address = address + offset;
data = Mem[address, 8, AccType\_NORMAL];
X[t] = data;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)[v29.05](#), AdvSIMD [v27.01](#)[v26.0](#), pseudocode [v85-xml-00bet8](#) [rc3](#)[v35.3](#); Build timestamp: [2018-09-13T13:20:04](#)[2018-06-16T09:04:45](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	1	1	0	0	0	0	1	0	imm9									0	1	Rn				Rt									
size										opc																									

### Post-index

```
LDRB <Wt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	1	0	imm9									1	1	Rn				Rt					
size										opc																					

### Pre-index

```
LDRB <Wt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
0		0		1		1		1		0		0		1		0		1		imm12												Rn				Rt			
size										opc																													

### Unsigned offset

```
LDRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRH \(immediate\)](#).

## Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;

```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```



```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

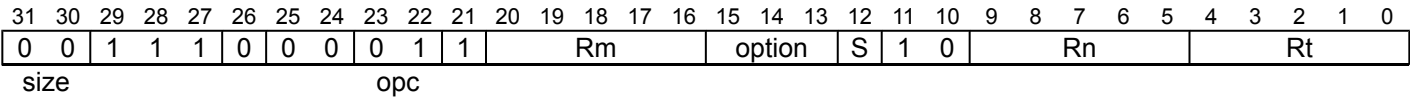
Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



### Extended register (option != 011)

```
LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### Shifted register (option == 011)

```
LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm>

When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm>

When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend>

Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

- <amount>

Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

bits(64) offset = ExtendReg(m, extend_type, shift);
if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding  UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	1	0	imm9									0	1	Rn				Rt									
size										opc																									

### Post-index

```
LDRH <Wt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	1	0	imm9									1	1	Rn				Rt									
size										opc																									

### Pre-index

```
LDRH <Wt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
0		1		1		1		0		0		1		0		1		imm12												Rn				Rt			
size										opc																											

### Unsigned offset

```
LDRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRH \(immediate\)](#).

## Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;

```





```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;        // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

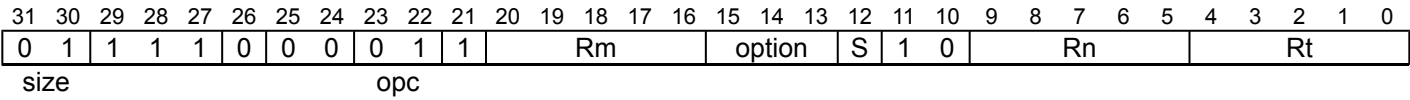
Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



32-bit

```
LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word indexif option<1> == '0' then
UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm>

When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm>

When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

- <amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

bits(64) offset = ExtendReg(m, extend_type, shift);
if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding  UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	1	1	0	0	0	1	x	0	imm9									0	1	Rn				Rt									
size										opc																									

### 32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>], #<simm>
```

### 64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	1	x	0	imm9									1	1	Rn				Rt					
size										opc																					

### 32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>, #<simm>]!
```

### 64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
0		0		1		1		1		0		0		1		1		x		imm12											Rn					Rt				
size										opc																														

### 32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>{, #<pimm>}]
```

### 64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRSB \(immediate\)](#).

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

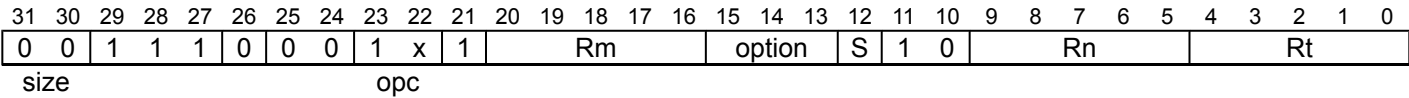
Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



32-bit with extended register offset (opc == 11 && option != 011)

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

32-bit with shifted register offset (opc == 11 && option == 011)

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

64-bit with extended register offset (opc == 10 && option != 011)

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

64-bit with shifted register offset (opc == 10 && option == 011)

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word indexif option<1> == '0' then
UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm>

When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm>

When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend>

Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

- <amount>

Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```





```

bits(64) offset = ExtendReg(m, extend_type, shift);
if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding  UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	1	x	0	imm9									0	1	Rn				Rt									
size										opc																									

#### 32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>], #<simm>
```

#### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	1	x	0	imm9									1	1	Rn				Rt									
size										opc																									

#### 32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, #<simm>]!
```

#### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
0		1		1		1		0		0		1		1		x		imm12													Rn								Rt			
size										opc																																

### 32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]
```

### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRSH \(immediate\)](#).

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
AccType acctype = AccType_NORMAL;  
MemOp memop;  
boolean signed;  
integer regsize;  
  
if opc<1> == '0' then  
    // store or zero-extending load  
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;  
    regsize = if size == '11' then 64 else 32;  
    signed = FALSE;  
else  
    if size == '11' then  
        UNDEFINED;  
    else  
        // sign-extending load  
        memop = if size == '11' then UnallocatedEncoding();  
    else  
        // sign-extending load  
        memop = MemOp_LOAD;  
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();  
        if size == '10' && opc<0> == '1' then UNDEFINED;  
    ();  
    regsize = if opc<0> == '1' then 32 else 64;  
    signed = TRUE;  
  
integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE;          // writeback is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE;           // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE;          // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding UNDEFINED;
    ();
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	x	1	Rm					option			S	1	0	Rn					Rt				
size								opc																							

### 32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount>	Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":
----------	---

S	<amount>
0	#0
1	#1



## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
        else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

bits(64) offset = ExtendReg(m, extend_type, shift);
if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding  UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	1	1	1	0	0	0	1	0	0	imm9									0	1	Rn				Rt									
size										opc																									

### Post-index

```
LDRSW <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	1	1	1	0	0	0	1	0	0	imm9									1	1	Rn				Rt									
size										opc																									

### Pre-index

```
LDRSW <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
1		0		1		1		1		0		0		1		1		0		imm12											Rn					Rt				
size										opc																														

### Unsigned offset

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRSW \(immediate\)](#).

## Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;

```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE;          // writeback is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE;           // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE;          // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding UNDEFINED;
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```



```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

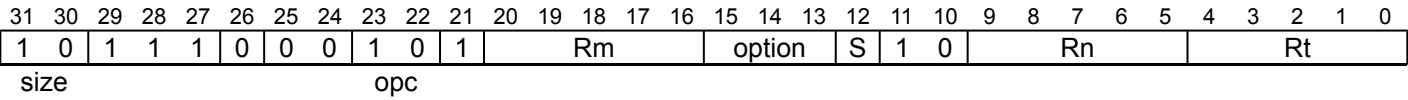
Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see [Load/Store addressing modes](#).



### 64-bit

```
LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word indexif option<1> == '0' then
UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

### Assembler Symbols

- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm>

When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm>

When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

- <amount>

Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

bits(64) offset = ExtendReg(m, extend_type, shift);
if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    e = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP       EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding  UNDEFINED;
        when Constraint_NOP       EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

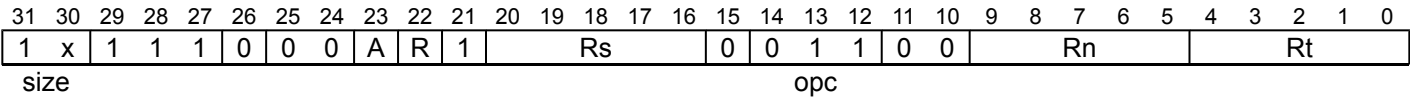
- If the destination register is not one of WZR or XZR, LDSETA and LDSETAL load from memory with acquire semantics.
- LDSETL and LDSETAL store to memory with release semantics.
- LDSET has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STSET, STSETL](#).

Integer  
(ARMv8.1)



### 32-bit LDSET (size == 10 && A == 0 && R == 0)

LDSET <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSETA (size == 10 && A == 1 && R == 0)

LDSETA <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSETAL (size == 10 && A == 1 && R == 1)

LDSETAL <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSETL (size == 10 && A == 0 && R == 1)

LDSETL <Ws>, <Wt>, [<Xn|SP>]

### 64-bit LDSET (size == 11 && A == 0 && R == 0)

LDSET <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSETA (size == 11 && A == 1 && R == 0)

LDSETA <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSETAL (size == 11 && A == 1 && R == 1)

LDSETAL <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSETL (size == 11 && A == 0 && R == 1)

LDSETL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.



<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSET, STSETL</a>	A == '0' && Rt == '11111'

Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25~~v29.05~~, AdvSIMD v27.01~~v26.0~~, pseudocode v85-xml-00~~bet8~~ rc3~~v35.3~~; Build timestamp: 2018-09-13T13~~2018-06-16T09~~:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- LDSETB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STSETB, STSETLB](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs				0	0	1	1	0	0	Rn				Rt						
size												opc																			

#### LDSETAB (A == 1 && R == 0)

```
LDSETAB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSETALB (A == 1 && R == 1)

```
LDSETALB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSETB (A == 0 && R == 0)

```
LDSETB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSETLB (A == 0 && R == 1)

```
LDSETLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = () then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSETB</a> , <a href="#">STSETLB</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)bet8\_rc3~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

## LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAH and LDSETALH load from memory with acquire semantics.
- LDSETLH and LDSETALH store to memory with release semantics.
- LDSETH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STSETH, STSETLH](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs				0	0	1	1	0	0	Rn				Rt						
size											opc																				

#### LDSETAH (A == 1 && R == 0)

```
LDSETAH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSETALH (A == 1 && R == 1)

```
LDSETALH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSETH (A == 0 && R == 0)

```
LDSETH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSETLH (A == 0 && R == 1)

```
LDSETLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = () then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSETH</a> , <a href="#">STSETLH</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)bet8\_rc3~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

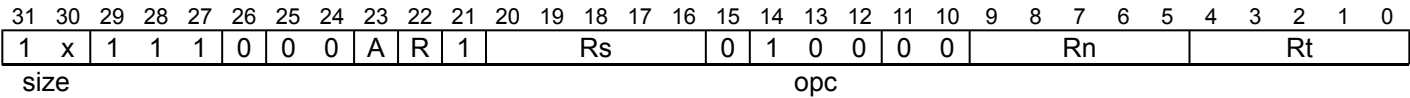
- If the destination register is not one of WZR or XZR, LDSMAXA and LDSMAXAL load from memory with acquire semantics.
- LDSMAXL and LDSMAXAL store to memory with release semantics.
- LDSMAX has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STSMAX, STSMAXL](#).

Integer  
(ARMv8.1)



### 32-bit LDSMAX (size == 10 && A == 0 && R == 0)

LDSMAX <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSMAXA (size == 10 && A == 1 && R == 0)

LDSMAXA <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSMAXAL (size == 10 && A == 1 && R == 1)

LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSMAXL (size == 10 && A == 0 && R == 1)

LDSMAXL <Ws>, <Wt>, [<Xn|SP>]

### 64-bit LDSMAX (size == 11 && A == 0 && R == 0)

LDSMAX <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSMAXA (size == 11 && A == 1 && R == 0)

LDSMAXA <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSMAXAL (size == 11 && A == 1 && R == 1)

LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSMAXL (size == 11 && A == 0 && R == 1)

LDSMAXL <Xs>, <Xt>, [<Xn|SP>]

```

if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;

```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

- <Xt>Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMAX, STSMAXL</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STSMAXB, STSMAXLB](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	1	0	0	0	0	Rn					Rt				
size											opc																				

#### LDSMAXAB (A == 1 && R == 0)

```
LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMAXALB (A == 1 && R == 1)

```
LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMAXB (A == 0 && R == 0)

```
LDSMAXB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMAXLB (A == 0 && R == 1)

```
LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMAXB, STSMAXB</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8](#) ~~rc3~~[v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

## LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- LDSMAXH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STSMAXH, STSMAXLH](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	1	0	0	0	0	Rn					Rt				
size											opc																				

#### LDSMAXAH (A == 1 && R == 0)

```
LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMAXALH (A == 1 && R == 1)

```
LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMAXH (A == 0 && R == 0)

```
LDSMAXH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMAXLH (A == 0 && R == 1)

```
LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMAXH</a> , <a href="#">STSMAXLH</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8](#) ~~rc3~~[v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

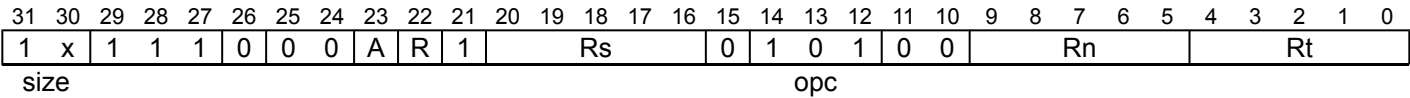
- If the destination register is not one of WZR or XZR, LDSMINA and LDSMINAL load from memory with acquire semantics.
- LDSMINL and LDSMINAL store to memory with release semantics.
- LDSMIN has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STSMIN, STSMINL](#).

Integer  
(ARMv8.1)



### 32-bit LDSMIN (size == 10 && A == 0 && R == 0)

LDSMIN <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSMINA (size == 10 && A == 1 && R == 0)

LDSMINA <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSMINAL (size == 10 && A == 1 && R == 1)

LDSMINAL <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSMINL (size == 10 && A == 0 && R == 1)

LDSMINL <Ws>, <Wt>, [<Xn|SP>]

### 64-bit LDSMIN (size == 11 && A == 0 && R == 0)

LDSMIN <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSMINA (size == 11 && A == 1 && R == 0)

LDSMINA <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSMINAL (size == 11 && A == 1 && R == 1)

LDSMINAL <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSMINL (size == 11 && A == 0 && R == 1)

LDSMINL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

- <Xt>Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMIN</a> , <a href="#">STSMINL</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25~~v29.05~~, AdvSIMD v27.01~~v26.0~~, pseudocode v85-xml-00~~bet8~~ rc3~~v35.3~~; Build timestamp: 2018-09-13T13~~2018-06-16T09~~:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

## LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STSMINB, STSMINLB](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	1	0	1	0	0	Rn					Rt				
size											opc																				

#### LDSMINAB (A == 1 && R == 0)

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMINALB (A == 1 && R == 1)

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMINB (A == 0 && R == 0)

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMINLB (A == 0 && R == 1)

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```



Assembler Symbols

- <Ws>
- Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt>
- Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>
- Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMINB</a> , <a href="#">STSMINLB</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- LDSMINH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STSMINH, STSMINLH](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	1	0	1	0	0	Rn					Rt				
size											opc																				

#### LDSMINAH (A == 1 && R == 0)

```
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMINALH (A == 1 && R == 1)

```
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMINH (A == 0 && R == 0)

```
LDSMINH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDSMINLH (A == 0 && R == 1)

```
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMINH</a> , <a href="#">STSMINLH</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)bet8\_rc3~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

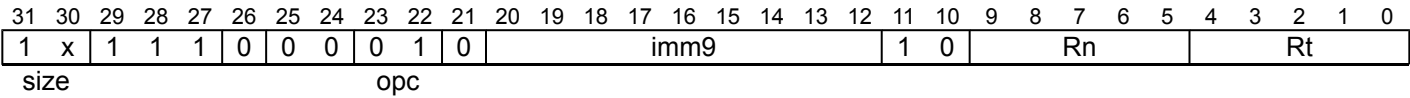
## LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR\_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



### 32-bit (size == 10)

```
LDTR <Wt>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (size == 11)

```
LDTR <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == (Rt); EL1AccType && !(acctype == EL2EnabledAccType_UNPRIV() &&
if ( HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = ( && HaveEL(EL2) && ) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
acctype = HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
acctype = AccType_UNPRIV;
else
acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = if size == '11' then 64 else 32;
signed = FALSE;
else
if size == '11' then
UNDEFINED;
else
// sign-extending load
memop = if size == '11' then UnallocatedEncoding();
else
// sign-extending load
memop = MemOp_LOAD;
if size == '10' && opc<0> == '1' then UnallocatedEncoding;
if size == '10' && opc<0> == '1' then UNDEFINED;
();

regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

integer datasize = 8 << scale;

```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



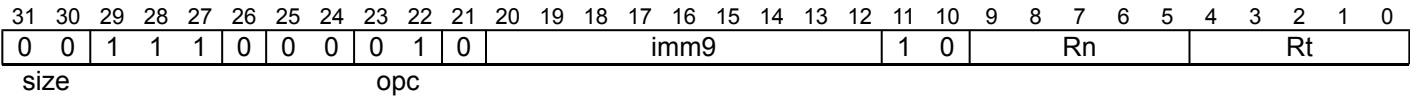
LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR\_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



Unscaled offset

```
LDTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == (Rt); EL1AccType && !(acctype == EL2EnabledAccType_UNPRIV() &&
if ( HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = ( && HaveEL(EL2) && ) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
acctype = HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
acctype = AccType_UNPRIV;
else
acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = if size == '11' then 64 else 32;
signed = FALSE;
else
if size == '11' then
UNDEFINED;
else
// sign-extending load
memop = if size == '11' then UnallocatedEncoding();
else
// sign-extending load
memop = MemOp_LOAD;
if size == '10' && opc<0> == '1' then UnallocatedEncoding;
if size == '10' && opc<0> == '1' then UNDEFINED;
();

regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

integer datasize = 8 << scale;

```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE;          // writeback is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE;           // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE;          // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding UNDEFINED;
    ();
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

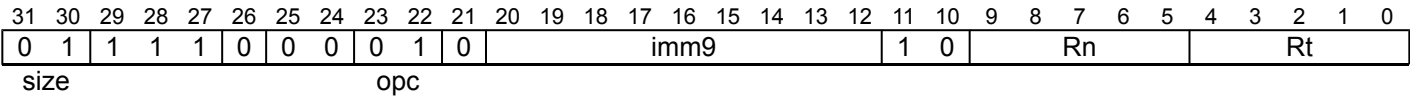
LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR\_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



Unscaled offset

```
LDTRH <Wt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == (Rt); EL1AccType && !(acctype == EL2EnabledAccType_UNPRIV() &&
if ( HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = ( && HaveEL(EL2) && ) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
acctype = HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
acctype = AccType_UNPRIV;
else
acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = if size == '11' then 64 else 32;
signed = FALSE;
else
if size == '11' then
UNDEFINED;
else
// sign-extending load
memop = if size == '11' then UnallocatedEncoding();
else
// sign-extending load
memop = MemOp_LOAD;
if size == '10' && opc<0> == '1' then UnallocatedEncoding;
if size == '10' && opc<0> == '1' then UNDEFINED;
();
regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

integer datasize = 8 << scale;

```





```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE;          // writeback is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE;           // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE;          // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding UNDEFINED;
    ();
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

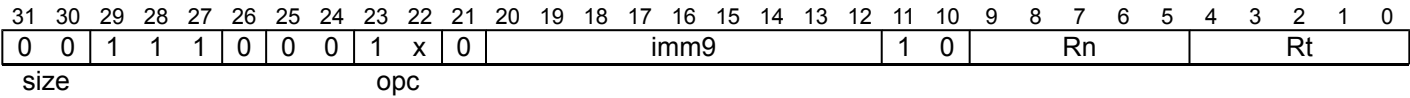
# LDTRSB

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR\_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



## 32-bit (opc == 11)

```
LDTRSB <Wt>, [<Xn|SP>{, #<sim>}]
```

## 64-bit (opc == 10)

```
LDTRSB <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == (Rt); EL1AccType && !(acctype == EL2EnabledAccType_UNPRIV() &&
if- ( HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = ( ) && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
acctype = HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
acctype = AccType_UNPRIV;
else
acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = if size == '11' then 64 else 32;
signed = FALSE;
else
if size == '11' then
UNDEFINED;
else
// sign-extending load
memop = if size == '11' then UnallocatedEncoding();
else
// sign-extending load
memop = MemOp_LOAD;
if size == '10' && opc<0> == '1' then UnallocatedEncoding;
if size == '10' && opc<0> == '1' then UNDEFINED;
( );

regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

integer datasize = 8 << scale;

```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

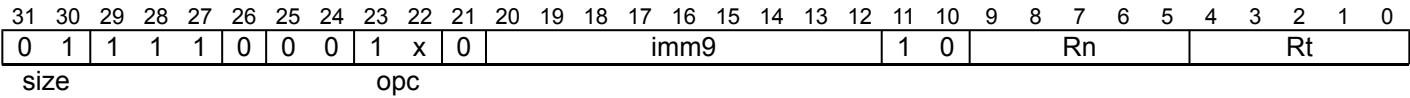
# LDTRSH

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR\_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



## 32-bit (opc == 11)

```
LDTRSH <Wt>, [<Xn|SP>{, #<sim>}]
```

## 64-bit (opc == 10)

```
LDTRSH <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.



## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == (Rt); EL1AccType && !(acctype == EL2EnabledAccType_UNPRIV() &&
if ( HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = ( && HaveEL(EL2) && ) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
acctype = HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
acctype = AccType_UNPRIV;
else
acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = if size == '11' then 64 else 32;
signed = FALSE;
else
if size == '11' then
UNDEFINED;
else
// sign-extending load
memop = if size == '11' then UnallocatedEncoding();
else
// sign-extending load
memop = MemOp_LOAD;
if size == '10' && opc<0> == '1' then UnallocatedEncoding;
if size == '10' && opc<0> == '1' then UNDEFINED;
();

regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

integer datasize = 8 << scale;

```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

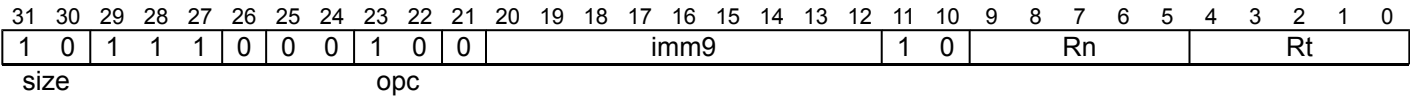
## LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR\_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



### Unscaled offset

```
LDTRSW <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == (Rt); EL1AccType && !(acctype == EL2EnabledAccType_UNPRIV() &&
if ( HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = ( && HaveEL(EL2) && ) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
acctype = HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
acctype = AccType_UNPRIV;
else
acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = if size == '11' then 64 else 32;
signed = FALSE;
else
if size == '11' then
UNDEFINED;
else
// sign-extending load
memop = if size == '11' then UnallocatedEncoding();
else
// sign-extending load
memop = MemOp_LOAD;
if size == '10' && opc<0> == '1' then UnallocatedEncoding;
if size == '10' && opc<0> == '1' then UNDEFINED;
();

regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

integer datasize = 8 << scale;

```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```



```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:20:09~~2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMAXA and LDUMAXAL load from memory with acquire semantics.
- LDUMAXL and LDUMAXAL store to memory with release semantics.
- LDUMAX has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STUMAX, STUMAXL](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	1	1	0	0	0	Rn				Rt						
size											opc																				

### 32-bit LDUMAX (size == 10 && A == 0 && R == 0)

LDUMAX <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDUMAXA (size == 10 && A == 1 && R == 0)

LDUMAXA <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDUMAXAL (size == 10 && A == 1 && R == 1)

LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDUMAXL (size == 10 && A == 0 && R == 1)

LDUMAXL <Ws>, <Wt>, [<Xn|SP>]

### 64-bit LDUMAX (size == 11 && A == 0 && R == 0)

LDUMAX <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDUMAXA (size == 11 && A == 1 && R == 0)

LDUMAXA <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDUMAXAL (size == 11 && A == 1 && R == 1)

LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDUMAXL (size == 11 && A == 0 && R == 1)

LDUMAXL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STUMAX, STUMAXL</a>	A == '0' && Rt == '11111'

Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDUMAXB, LDUMAXB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXB and LDUMAXALB load from memory with acquire semantics.
- LDUMAXLB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STUMAXB, STUMAXLB](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	1	1	0	0	0	Rn					Rt				
size											opc																				

#### LDUMAXB (A == 1 && R == 0)

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMAXALB (A == 1 && R == 1)

```
LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMAXB (A == 0 && R == 0)

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMAXLB (A == 0 && R == 1)

```
LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STUMAXB, STUMAXLB</a>	<code>A == '0' &amp;&amp; Rt == '11111'</code>

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa `v30.25v29.05`, AdvSIMD `v27.01v26.0`, pseudocode `v85-xml-00bet8_rc3v35.3`; Build timestamp: `2018-09-13T13:2018-06-16T09:0445`

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXLH and LDUMAXALH store to memory with release semantics.
- LDUMAXH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STUMAXH, STUMAXLH](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	1	1	0	0	0	Rn					Rt				
size											opc																				

#### LDUMAXAH (A == 1 && R == 0)

```
LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMAXALH (A == 1 && R == 1)

```
LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMAXH (A == 0 && R == 0)

```
LDUMAXH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMAXLH (A == 0 && R == 1)

```
LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType ORDEREDATOMICRW else AccType ORDEREDRW;
AccType stacctype = if R == '1' then AccType ORDEREDATOMICRW else AccType ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STUMAXH</a> , <a href="#">STUMAXLH</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)bet8\_rc3~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)



LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

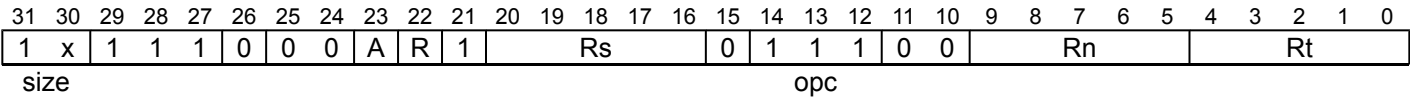
- If the destination register is not one of WZR or XZR, LDUMINA and LDUMINAL load from memory with acquire semantics.
- LDUMINL and LDUMINAL store to memory with release semantics.
- LDUMIN has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STUMIN, STUMINL](#).

Integer  
(ARMv8.1)



### 32-bit LDUMIN (size == 10 && A == 0 && R == 0)

LDUMIN <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDUMINA (size == 10 && A == 1 && R == 0)

LDUMINA <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDUMINAL (size == 10 && A == 1 && R == 1)

LDUMINAL <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDUMINL (size == 10 && A == 0 && R == 1)

LDUMINL <Ws>, <Wt>, [<Xn|SP>]

### 64-bit LDUMIN (size == 11 && A == 0 && R == 0)

LDUMIN <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDUMINA (size == 11 && A == 1 && R == 0)

LDUMINA <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDUMINAL (size == 11 && A == 1 && R == 1)

LDUMINAL <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDUMINL (size == 11 && A == 0 && R == 1)

LDUMINL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW AccType_ORDEREDRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

- <Xt>Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STUMIN, STUMINL</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

## LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAB and LDUMINALB load from memory with acquire semantics.
- LDUMINLB and LDUMINALB store to memory with release semantics.
- LDUMINB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STUMINB, STUMINLB](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	1	1	1	0	0	Rn					Rt				
size											opc																				

#### LDUMINAB (A == 1 && R == 0)

```
LDUMINAB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMINALB (A == 1 && R == 1)

```
LDUMINALB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMINB (A == 0 && R == 0)

```
LDUMINB <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMINLB (A == 0 && R == 1)

```
LDUMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STUMINB</a> , <a href="#">STUMINLB</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25~~v29.05~~, AdvSIMD v27.01~~v26.0~~, pseudocode v85-xml-00bet8 rc3~~v35.3~~; Build timestamp: 2018-09-13T13~~2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINALH store to memory with release semantics.
- LDUMINH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias [STUMINH, STUMINLH](#).

### Integer (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	1	1	1	0	0	Rn					Rt				
size											opc																				

#### LDUMINAH (A == 1 && R == 0)

```
LDUMINAH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMINALH (A == 1 && R == 1)

```
LDUMINALH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMINH (A == 0 && R == 0)

```
LDUMINH <Ws>, <Wt>, [<Xn|SP>]
```

#### LDUMINLH (A == 0 && R == 1)

```
LDUMINLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
AccType stacctype = if R == '1' then AccType ORDEREDATOMICRW AccType ORDEREDRW else AccType ATOMICRW;
MemAtomicOp op;
case opc of
  when '000' op = MemAtomicOp_ADD;
  when '001' op = MemAtomicOp_BIC;
  when '010' op = MemAtomicOp_EOR;
  when '011' op = MemAtomicOp_ORR;
  when '100' op = MemAtomicOp_SMAX;
  when '101' op = MemAtomicOp_SMIN;
  when '110' op = MemAtomicOp_UMAX;
  when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STUMINH</a> , <a href="#">STUMINLH</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

if value == HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

case op of
    when MemAtomicOp_ADD    result = data + value;
    when MemAtomicOp_BIC    result = data AND NOT(value);
    when MemAtomicOp_EOR    result = data EOR value;
    when MemAtomicOp_ORR    result = data OR value;
    when MemAtomicOp_SMAX   result = if SInt(data) > SInt(value) then data else value;
    when MemAtomicOp_SMIN   result = if SInt(data) > SInt(value) then value else data;
    when MemAtomicOp_UMAX   result = if UInt(data) > UInt(value) then data else value;
    when MemAtomicOp_UMIN   result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)bet8\_rc3~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

## LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	1	0	imm9									0	0	Rn				Rt					
opc																															

### 8-bit (size == 00 && opc == 01)

```
LDUR <Bt>, [<Xn|SP>{, #<sim>}]
```

### 16-bit (size == 01 && opc == 01)

```
LDUR <Ht>, [<Xn|SP>{, #<sim>}]
```

### 32-bit (size == 10 && opc == 01)

```
LDUR <St>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (size == 11 && opc == 01)

```
LDUR <Dt>, [<Xn|SP>{, #<sim>}]
```

### 128-bit (size == 00 && opc == 11)

```
LDUR <Qt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>.size);
if scale > 4 then UNDEFINED;
bits(64) offset = if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```



## Operation

```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

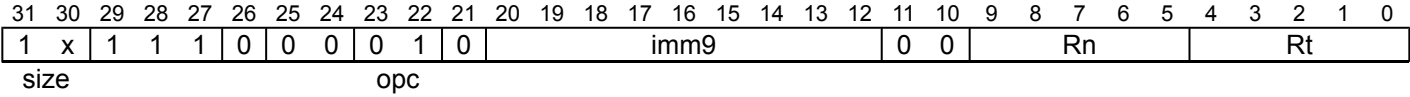
(old)

htmldiff from-

(new)

# LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



## 32-bit (size == 10)

```
LDUR <Wt>, [<Xn|SP>{, #<sim>}]
```

## 64-bit (size == 11)

```
LDUR <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	1	0	imm9									0	0	Rn			Rt						
size											opc																				

### Unscaled offset

```
LDURB <Wt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```





```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	1	0	imm9										0	0	Rn				Rt								
size											opc																								

### Unscaled offset

```
LDURH <Wt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE;          // writeback is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE;           // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE;          // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding UNDEFINED;
    ();
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

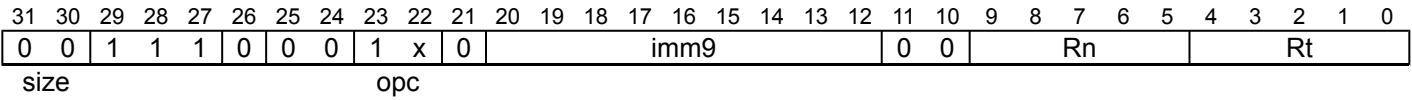
Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



32-bit (opc == 11)

```
LDURSB <Wt>, [<Xn|SP>{, #<sim>}]
```

64-bit (opc == 10)

```
LDURSB <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```





```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

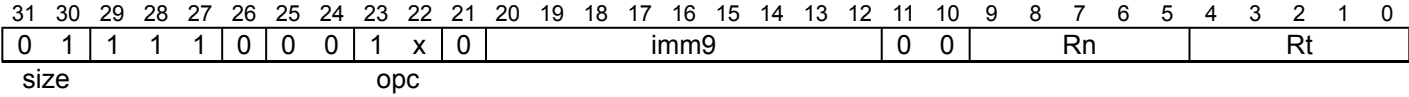
Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



## 32-bit (opc == 11)

```
LDURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

## 64-bit (opc == 10)

```
LDURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	0	1	1	1	0	0	0	1	0	0	imm9									0	0	Rn			Rt											
size											opc																									

### Unscaled offset

```
LDURSW <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

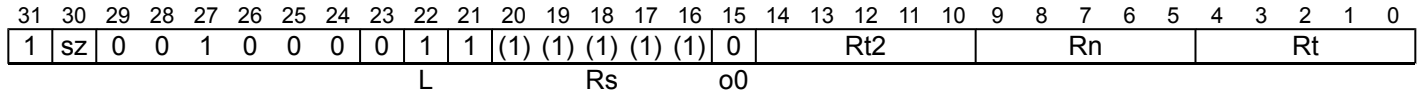
Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). For information about memory accesses see [Load/Store addressing modes](#).



### 32-bit (sz == 0)

```
LDXP <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

### 64-bit (sz == 1)

```
LDXP <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMICAccType_ORDERED else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDXP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

    when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

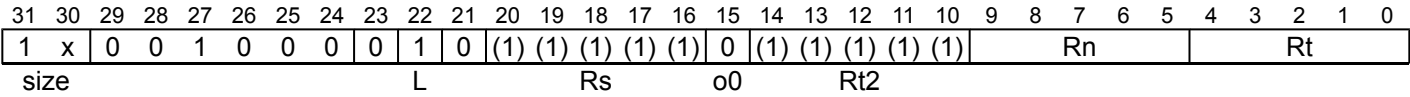
htmldiff from-

(new)



LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.



32-bit (size == 10)

```
LDXR <Wt>, [<Xn|SP>{, #0}]
```

64-bit (size == 11)

```
LDXR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF        UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP          EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF        UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP          EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF        UnallocatedEncoding UNDEFINED;
            when Constraint_NOP          EndOfInstruction();

    when Constraint_NOP          EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size								L			Rs				o0		Rt2														

### No offset

```
LDXRB <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType ORDEREDATOMIC else AccType ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF        UNDEFINED;
        when UnallocatedEncoding()
        when Constraint_NOP          EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF        UNDEFINED;
            when UnallocatedEncoding()
            when Constraint_NOP          EndOfInstruction();
        if s == n && n != 31 then
            Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
            assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
            case c of
                when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
                when Constraint_NONE        rn_unknown = FALSE;   // address is original base
                when Constraint_UNDEF        UnallocatedEncoding() UNDEFINED;
                when Constraint_NOP          EndOfInstruction();

        ());

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size								L			Rs				o0		Rt2														

### No offset

```
LDXRH <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType ORDEREDATOMIC else AccType ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

() ;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

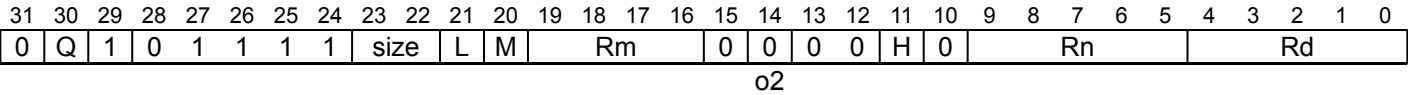
htmldiff from-

(new)

### MLA (by element)

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Vector

```
MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
integer d = otherwise UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

### Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

- Restricted to V0-V15 when element size <Ts> is H.
- <Ts> Is an element size specifier, encoded in “size”:

<index>	Is the element index, encoded in “size:L:H:M”:
---------	--

## Operation

## Operational information

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

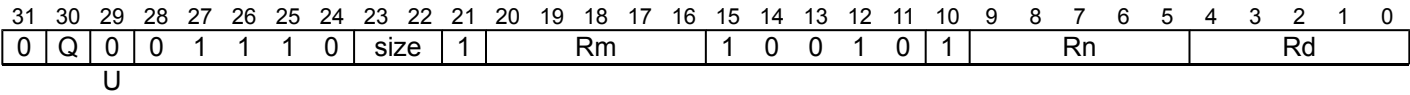
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

### MLA (by element)

MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Three registers of the same type

```
MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

Assembler Symbols

- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1) * UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

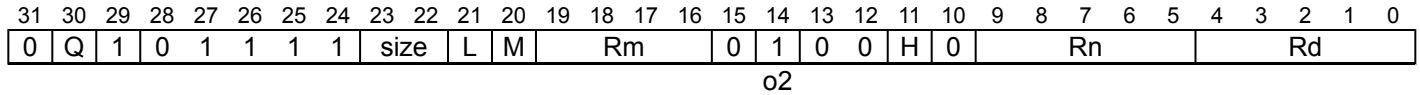
(old)	htmldiff from-	(new)
-------	----------------	-------



## MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Vector

MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UNDEFINED;
integer d = otherwise UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

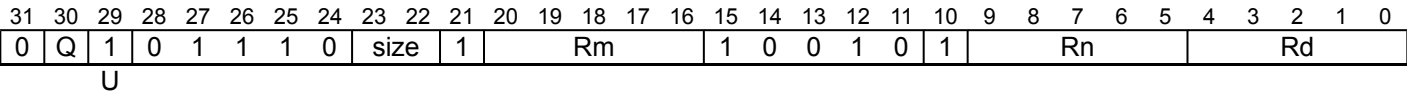
htmldiff from-

(new)

MLS (vector)

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Three registers of the same type

```
MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

Assembler Symbols

- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1) * UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:18+00:002018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

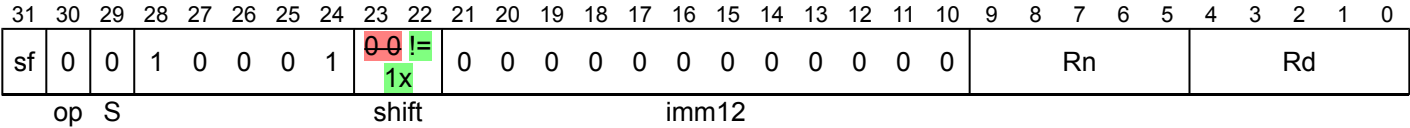
(old)	htmldiff from-	(new)
-------	----------------	-------

## MOV (to/from SP)

Move between register and stack pointer: Rd = Rn.

This is an alias of [ADD \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [ADD \(immediate\)](#).
- The description of [ADD \(immediate\)](#) gives the operational pseudocode for this instruction.



### 32-bit (sf == 0)

MOV <Wd|WSP>, <Wn|WSP>

is equivalent to

ADD <Wd|WSP>, <Wn|WSP>, #0

and is the preferred disassembly when (Rd == '11111' || Rn == '11111').

### 64-bit (sf == 1)

MOV <Xd|SP>, <Xn|SP>

is equivalent to

ADD <Xd|SP>, <Xn|SP>, #0

and is the preferred disassembly when (Rd == '11111' || Rn == '11111').

## Assembler Symbols

<Wd WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [ADD \(immediate\)](#) gives the operational pseudocode for this instruction.

MOVI

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD&FP register. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode				0	1	d	e	f	g	h	Rd				

### 8-bit (op == 0 && cmode == 1110)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #0}
```

### 16-bit shifted immediate (op == 0 && cmode == 10x0)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit shifted immediate (op == 0 && cmode == 0xx0)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit shifting ones (op == 0 && cmode == 110x)

```
MOVI <Vd>.<T>, #<imm8>, MSL #<amount>
```

### 64-bit scalar (Q == 0 && op == 1 && cmode == 1110)

```
MOVI <Dd>, #<imm>
```

### 64-bit vector (Q == 1 && op == 1 && cmode == 1110)

```
MOVI <Vd>.2D, #<imm>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11110' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = if Q == '0' then UnallocatedEncoding();
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<imm>	Is a 64-bit immediate 'aaaaaaaabbbbbbbcccccccddeeeeeeffffffffggggggghhhhhhhh', encoded in "a:b:c:d:e:f:g:h".
<T>	For the 8-bit variant: is an arrangement specifier, encoded in "Q".

Q	<T>
0	8B
1	16B

For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



(old)

htmldiff from-

(new)

## MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sf		1		1		1		0		0		1		0		1		hw		imm16																		Rd			
opc																																									

### 32-bit (sf == 0)

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

### 64-bit (sf == 1)

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
bits(16) imm = imm16;
integer pos;
MoveWideOp opcode;

case opc of
  when '00' opcode = MoveWideOp_N;
  when '10' opcode = MoveWideOp_Z;
  when '11' opcode = MoveWideOp_K;
  otherwise UNDEFINED;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = otherwise UnallocatedEncoding();

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

## Operation

```
bits(datasize) result;

if opcode == MoveWideOp_K then
  result = X[d];
else
  result = Zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N then
  result = NOT(result);
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

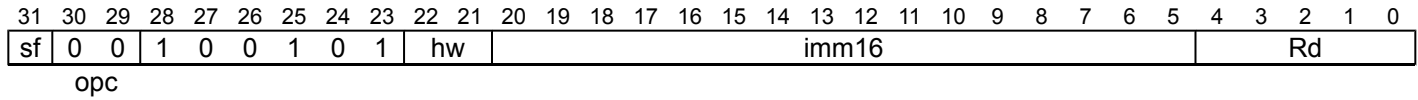
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## MOVN

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias [MOV \(inverted wide immediate\)](#).



### 32-bit (sf == 0)

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

### 64-bit (sf == 1)

```
MOVN <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
bits(16) imm = imm16;
integer pos;
MoveWideOp opcode;

case opc of
  when '00' opcode = MoveWideOp_N;
  when '10' opcode = MoveWideOp_Z;
  when '11' opcode = MoveWideOp_K;
  otherwise UNDEFINED;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = otherwise UnallocatedEncoding();

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt (hw: '0000');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

## Alias Conditions

Alias	Of variant Is preferred when	
<a href="#">MOV (inverted wide immediate)</a>	64-bit	! (IsZero(imm16) && hw != '00')
<a href="#">MOV (inverted wide immediate)</a>	32-bit	! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)

Operation

```
bits(datasize) result;

if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N then
    result = NOT(result);
X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

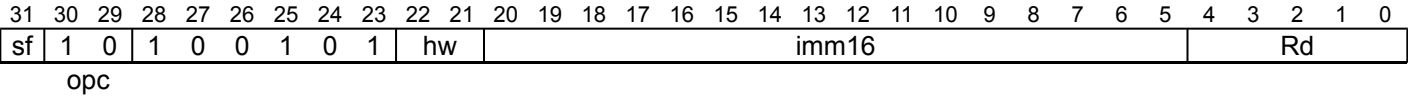
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# MOVZ

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias [MOV \(wide immediate\)](#).



## 32-bit (sf == 0)

```
MOVZ <Wd>, #<imm>{, LSL #<shift>}
```

## 64-bit (sf == 1)

```
MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
bits(16) imm = imm16;
integer pos;
MoveWideOp opcode;

case opc of
  when '00' opcode = MoveWideOp_N;
  when '10' opcode = MoveWideOp_Z;
  when '11' opcode = MoveWideOp_K;
  otherwise UNDEFINED;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = otherwise UnallocatedEncoding();

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt (hw: '0000');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

## Alias Conditions

Alias	Is preferred when
<a href="#">MOV (wide immediate)</a>	<code>! (IsZero(imm16) &amp;&amp; hw != '00')</code>

Operation

```
bits(datasize) result;

if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N then
    result = NOT(result);
X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see [Process state, PSTATE](#).

The bits that can be written by are this D, instruction A, are I, F, and SP. This set of bits is expanded in extensions to the architecture as follows:

- [ARMv8.1-PAN](#) PSTATE.D, adds PSTATE.A, the PSTATE.I, PAN, PSTATE.F, and PSTATE.SP bit.
- [ARMv8.2-UAO](#) If [ARMv8.0-SSBS](#) is added implemented, the PSTATE.SSBS.UAO bit.
- [ARMv8.4-DIT](#) If [ARMv8.1-PAN](#) is added implemented, the PSTATE.PAN.DIT bit.
- If [ARMv8.2-UAO](#) is implemented, PSTATE.UAO.
- If [ARMv8.4-DIT](#) is implemented, PSTATE.DIT.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0		op1		0	1	0	0			CRm			op2		1	1	1	1	1

## System

MSR <pstatefield>, #<imm>

```

if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFlag";
if op1 == '000' && op2 == '010' then SEE "AXFlag";

AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');

bits(4) operand = CRm;
PSTATEField field;
case op1:op2 of
    when '000 000'
        SEE "CFINV";
    when '000 011'
        if !HaveUAOExt() then
            UNDEFINED;
        field =() then UnallocatedEncoding();
        field = PSTATEField_UAO;
    when '000 100'
        if !HavePANExt() then
            UNDEFINED;
        field =() then UnallocatedEncoding();
        field = PSTATEField_PAN;
    when '000 101' field = PSTATEField_SP;
    when '011 010'
        if !HaveDITExt() then
            UNDEFINED;
        field =() then UnallocatedEncoding();
        field = PSTATEField_DIT;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    when '011 001'
        if !otherwiseHaveSSBSExtUnallocatedEncoding() then
            UNDEFINED;
        field = PSTATEField_SSBS;
        otherwise UNDEFINED;
();

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if op1 == '011' && PSTATE.EL == ELO && (IsInHost() || SCTLR_EL1.UMA == '0') then
    AArch64.SystemRegisterTrap(EL1, '00', op2, op1, '0100', '11111', CRm, '0');

```

## Assembler Symbols

<pstatefield> Is a PSTATE field name, encoded in “op1:op2”:



op1	op2	<pstatefield>	Architectural Feature
000	00x011	UAOSEE PSTATE	-ARMv8.2-UAO
000	010100	PANSEE PSTATE	-ARMv8.1-PAN
000	011101	UAO SPSEL	ARMv8.2-UAO-
000011	100010	PAN DIT	ARMv8.1-PANARMv8.4-DIT
000011	101110	SPSEL DAIFSet	-
000011	11x111	RESERVED DAIFClr	-
001	xxx	RESERVED	-
010	xxx	RESERVED	-
011	000	RESERVED	-
011	001	SSBS	ARMv8.0-SpecRest
011	010	DIT	ARMv8.4-DIT
011	011	RESERVED	-
011	10x	RESERVED	-
011	110	DAIFSet	-
011	111	DAIFClr	-
1xx	xxx	RESERVED	-

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

## Operation

```

case field of
  when PSTATEField SSBS
    PSTATE.SSBS = operand<0>;
  when PSTATEField SP
    PSTATE.SP = operand<0>;
  when PSTATEField DAIFSet
    PSTATE.D = PSTATE.D OR operand<3>;
    PSTATE.A = PSTATE.A OR operand<2>;
    PSTATE.I = PSTATE.I OR operand<1>;
    PSTATE.F = PSTATE.F OR operand<0>;
  when PSTATEField DAIFClr
    PSTATE.D = PSTATE.D AND NOT(operand<3>);
    PSTATE.A = PSTATE.A AND NOT(operand<2>);
    PSTATE.I = PSTATE.I AND NOT(operand<1>);
    PSTATE.F = PSTATE.F AND NOT(operand<0>);
  when PSTATEField PAN
    PSTATE.PAN = operand<0>;
  when PSTATEField UAO
    PSTATE.UAO = operand<0>;
  when PSTATEField DIT
    PSTATE.DIT = operand<0>;

```

Internal version only: isa v30.25-v29.05, AdvSIMD v27.01-v26.0, pseudocode v85-xml-00bet8\_rc3-v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## MUL (by element)

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M	Rm				1	0	0	0	H	0	Rn				Rd						

### Vector

MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
integer d = otherwise UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

&lt;index&gt;

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

checkFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxsizesize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:092018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

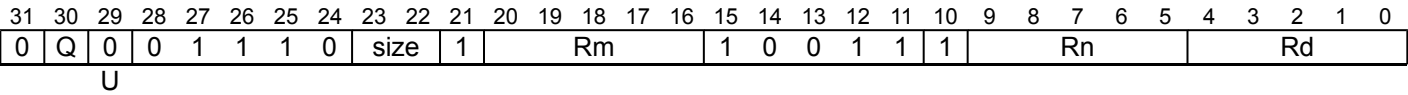
## htmldiff from-

(new)

## MUL (vector)

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

```
MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << if U == '1' && size != '00' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

### Assembler Symbols

- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1) * UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## MVNI

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode				0	1	d	e	f	g	h	Rd				
op																															

### 16-bit shifted immediate (cmode == 10x0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit shifted immediate (cmode == 0xx0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit shifting ones (cmode == 110x)

```
MVNI <Vd>.<T>, #<imm8>, MSL #<amount>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp MOVI;
  when '0xx01' operation = ImmediateOp MVNI;
  when '0xx10' operation = ImmediateOp ORR;
  when '0xx11' operation = ImmediateOp BIC;
  when '10x00' operation = ImmediateOp MOVI;
  when '10x01' operation = ImmediateOp MVNI;
  when '10x10' operation = ImmediateOp ORR;
  when '10x11' operation = ImmediateOp BIC;
  when '110x0' operation = ImmediateOp MOVI;
  when '110x1' operation = ImmediateOp MVNI;
  when '1110x' operation = ImmediateOp MOVI;
  when '11110' operation = ImmediateOp MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = if Q == '0' then UnallocatedEncoding();
    operation = ImmediateOp MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa [v30.25-v29.05](#), AdvSIMD [v27.01-v26.0](#), pseudocode [v85-xml-00bet8\\_rc3-v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

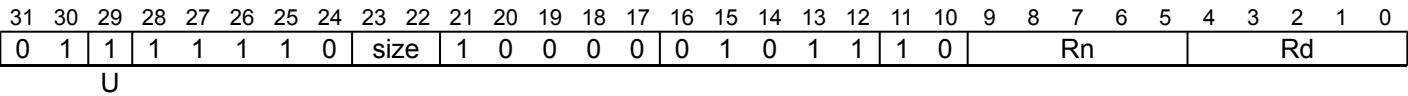
## NEG (vector)

Negate (vector). This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar



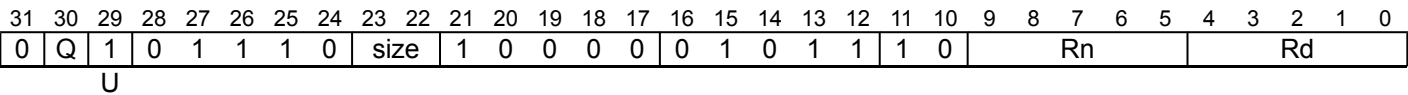
### Scalar

NEG <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

### Vector



### Vector

NEG <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

### Assembler Symbols

<V>	Is a width specifier, encoded in “size”:								
	<table><tr><th>size</th><th>&lt;V&gt;</th></tr><tr><td>0x</td><td>RESERVED</td></tr><tr><td>10</td><td>RESERVED</td></tr><tr><td>11</td><td>D</td></tr></table>	size	<V>	0x	RESERVED	10	RESERVED	11	D
size	<V>								
0x	RESERVED								
10	RESERVED								
11	D								
<d>	Is the number of the SIMD&FP destination register, encoded in the "Rd" field.								
<n>	Is the number of the SIMD&FP source register, encoded in the "Rn" field.								



**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>** Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
------	--

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1
																CRm				op2											

## System

NOP

```

SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111'
    SEE "XPACLRI";
  when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 100'
    op = SystemHintOp_CSDB;
  when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0'
    op = otherwise SystemHintOp_BTI;
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction(); // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

## Operational information

If [PSTATE.DIT](#) is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

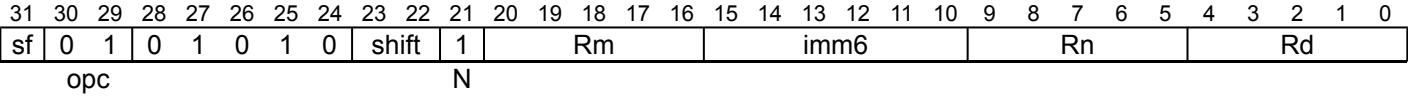
htmldiff from-

(new)

## ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias [MVN](#).



### 32-bit (sf == 0)

ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

ORN <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then UNDEFINED; if sf == '0' && imm6<5> == '1' then
ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

## Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Alias Conditions

Alias	Is preferred when
<a href="#">MVN</a>	Rn == '11111'

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR  result = operand1 OR  operand2;
  when LogicalOp_EOR  result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25~~v29.05~~, AdvSIMD v27.01~~v26.0~~, pseudocode v85-xml-00~~bet8\_rc3~~v35.3; Build timestamp: 2018-09-13T13~~2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd				
op												cmode																			

### 16-bit (cmode == 10x1)

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit (cmode == 0xx1)

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11110' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = if Q == '0' then UnallocatedEncoding();
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

&lt;imm8&gt;

&lt;amount&gt;

<b>cmode&lt;1&gt;</b>	<b>&lt;amount&gt;</b>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;

```

## Operational information

If `PSTATE.DIT` is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:092018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

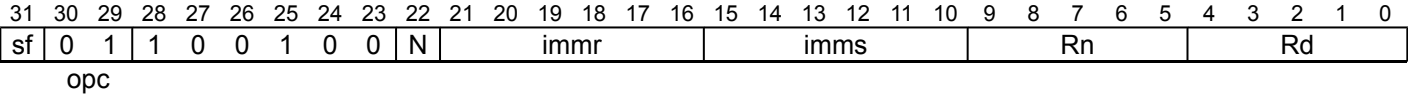
## htmldiff from-

(new)

## ORR (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias [MOV \(bitmask immediate\)](#).



### 32-bit (sf == 0 && N == 0)

```
ORR <Wd|WSP>, <Wn>, #<imm>
```

### 64-bit (sf == 1)

```
ORR <Xd|SP>, <Xn>, #<imm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) =if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

## Assembler Symbols

<Wd WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

## Alias Conditions

Alias	Is preferred when
<a href="#">MOV (bitmask immediate)</a>	Rn == '11111' && ! <a href="#">MoveWidePreferred</a> (sf, N, imms, immr)



Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR result = operand1 OR operand2;
  when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## ORR (shifted register)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias [MOV \(register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
sf	0	1	0	1	0	1	0	shift	0	Rm						imm6						Rn						Rd					
opc								N																									

### 32-bit (sf == 0)

```
ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
  when '00' op = LogicalOp_AND; setflags = FALSE;
  when '01' op = LogicalOp_ORR; setflags = FALSE;
  when '10' op = LogicalOp_EOR; setflags = FALSE;
  when '11' op = LogicalOp_AND; setflags = TRUE;

if sf == '0' && imm6<5> == '1' then UNDEFINED; if sf == '0' && imm6<5> == '1' then
ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
----------	--

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (register)</a>	shift == '00' && imm6 == '000000' && Rn == '11111'

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

if invert then operand2 = NOT(operand2);

case op of
  when LogicalOp_AND result = operand1 AND operand2;
  when LogicalOp_ORR  result = operand1 OR  operand2;
  when LogicalOp_EOR  result = operand1 EOR operand2;

if setflags then
  PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## PACDA, PACDZA

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDA.
- The value zero, for PACDZA.

### Integer (ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	0	Rn				Rd					

#### PACDA (Z == 0)

PACDA <Xd>, <Xn|SP>

#### PACDZA (Z == 1 && Rn == 11111)

PACDZA <Xd>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then UNDEFINED;() thenUnallocatedEncoding();

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then UnallocatedEncoding();
```

### Assembler Symbols

- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

### Operation

```
if source_is_sp then
    X[d] = AddPACDA(X[d], SP[]);
else
    X[d] = AddPACDA(X[d], X[n]);
```

## PACDB, PACDZB

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDB.
- The value zero, for PACDZB.

### Integer (ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	1	Rn				Rd					

#### PACDB (Z == 0)

PACDB <Xd>, <Xn|SP>

#### PACDZB (Z == 1 && Rn == 11111)

PACDZB <Xd>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then UNDEFINED;() thenUnallocatedEncoding();

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then UnallocatedEncoding();
```

### Assembler Symbols

- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

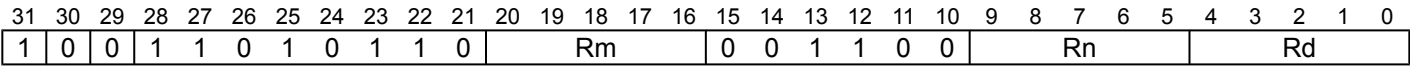
### Operation

```
if source_is_sp then
    X[d] = AddPACDB(X[d], SP[]);
else
    X[d] = AddPACDB(X[d], X[n]);
```

PACGA

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of the destination register.

Integer  
(ARMv8.3)



Integer

PACGA <Xd>, <Xn>, <Xm|SP>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if !HavePACExt() then
    UNDEFINED;

if m == 31 then source_is_sp = TRUE; () then UnallocatedEncoding();
if m == 31 then source_is_sp = TRUE;
```

Assembler Symbols

- <Xd>Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm|SP>Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

Operation

```
if source_is_sp then
    X[d] = AddPACGA(X[n], SP[]);
else
    X[d] = AddPACGA(X[n], X[m]);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by <Xd> for PACIA and PACIZA.
- In X17, for PACIA1716.
- In X30, for PACIASP and PACIAZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIA.
- The value zero, for PACIZA and PACIAZ.
- In X16, for PACIA1716.
- In SP, for PACIASP.

It has encodings from 2 classes: [Integer](#) and [System](#)

### Integer (ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	0	0	Rn				Rd					

### PACIA (Z == 0)

PACIA <Xd>, <Xn|SP>

### PACIZA (Z == 1 && Rn == 11111)

PACIZA <Xd>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE;
else // PACIZA
    if n != 31 then UNDEFINED; () then UnallocatedEncoding();

if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE;
else // PACIZA
    if n != 31 then UnallocatedEncoding();
```

### System (ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	0	x	1	1	1	1	1
												CRm																op2			

**PACIA1716 (CRm == 0001 && op2 == 000)**

PACIA1716

**PACIASP (CRm == 0011 && op2 == 001)**

PACIASP

**PACIAZ (CRm == 0011 && op2 == 000)**

PACIAZ

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 000' // PACIAZ
    d = 30;
    n = 31;
  when '0011 001' // PACIASP
    d = 30;
    source_is_sp = TRUE;
  if when '0001 000' // PACIA1716
    d = 17;
    n = 16;
  when '0001 010' SEE "PACIB";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 01x' SEE "PACIB";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPAQLRI"; HaveBTIExt() then
    BTypeCompatible = BTypeCompatible PACIXSP();
  when '0001 000' // PACIA1716
    d = 17;
    n = 16;
  when '0001 010' SEE "PACIB";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 01x' SEE "PACIB";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPAQLRI";
```

## Assembler Symbols

**<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

## Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AddPACIA(X[d], SP[]);
    else
        X[d] = AddPACIA(X[d], X[n]);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)



## PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by <Xd> for PACIB and PACIZB.
- In X17, for PACIB1716.
- In X30, for PACIBSP and PACIBZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIB.
- The value zero, for PACIZB and PACIBZ.
- In X16, for PACIB1716.
- In SP, for PACIBSP.

It has encodings from 2 classes: [Integer](#) and [System](#)

### Integer (ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	0	1	Rn				Rd					

### PACIB (Z == 0)

PACIB <Xd>, <Xn|SP>

### PACIZB (Z == 1 && Rn == 11111)

PACIZB <Xd>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then UNDEFINED; () then UnallocatedEncoding();

if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then UnallocatedEncoding();
```

### System (ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	1	x	1	1	1	1	1
												CRm				op2															

**PACIB1716 (CRm == 0001 && op2 == 010)**

PACIB1716

**PACIBSP (CRm == 0011 && op2 == 011)**

PACIBSP

**PACIBZ (CRm == 0011 && op2 == 010)**

PACIBZ

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 010' // PACIBZ
    d = 30;
    n = 31;
  when '0011 011' // PACIBSP
    d = 30;
    source_is_sp = TRUE;
  if when '0001 010' // PACIB1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 00x' SEE "PACIA";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPAQLRI"; HaveBTIExt() then
    BTypeCompatible = BTypeCompatible PACIXSP();
  when '0001 010' // PACIB1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 00x' SEE "PACIA";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPAQLRI";
```

## Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

## Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AddPACIB(X[d], SP[]);
    else
        X[d] = AddPACIB(X[d], X[n]);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:44Z2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

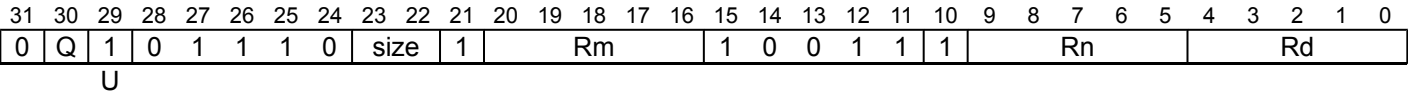
(old)	htmldiff from-	(new)

PMUL

Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Three registers of the same type

```
PMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << if U == '1' && size != '00' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

Assembler Symbols

- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

- <Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1) * UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25**~~v29.05~~, AdvSIMD **v27.01**~~v26.0~~, pseudocode **v85-xml-00**~~bet8\_rc3~~**v35.3**; Build timestamp: **2018-09-13T13:20:18-06:16**~~T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## PMULL, PMULL2

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}*.

The PMULL instruction extracts each source vector from the lower half of each source register, while the PMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			1	1	1	0	0	0				Rn				Rd		

### Three registers, not all the same type

```
PMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '01' || size == '10' then UNDEFINED;
if size == '11' && !if size == '01' || size == '10' then ReservedValue();
if size == '11' && !HaveBit128PMULLExt() then UnallocatedEncoding() then UNDEFINED;
();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	RESERVED
10	RESERVED
11	1Q

The ‘1Q’ arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	x	RESERVED
10	x	RESERVED
11	0	1D
11	1	2D

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, 2*esize] = PolynomialMult(element1, element2);

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see [Prefetch memory](#).

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	imm12												Rn				Rt					
size										opc																					

### Unsigned offset

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

### Assembler Symbols

<prfop>	Is the prefetch operation, defined as <type><target><policy>. <type> is one of:
<b>PLD</b>	Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
<b>PLI</b>	Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
<b>PST</b>	Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.
	<target> is one of:
<b>L1</b>	Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
<b>L2</b>	Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
<b>L3</b>	Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.
	<policy> is one of:
<b>KEEP</b>	Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
<b>STRM</b>	Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.
	For more information on these prefetch operations, see <a href="#">Prefetch memory</a> . For other encodings of the "Rt" field, use <imm5>.
<imm5>	Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```





```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see [Prefetch memory](#).

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	Rm				option			S	1	0	Rn				Rt						
size											opc																				

## Integer

```
PRFM (<prfop>|<#imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>. <type> is one of:

### PLD

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

### PLI

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

### PST

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

### L1

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

### L2

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

### L3

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

### KEEP

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

### STRM

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see [Prefetch memory](#).

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

bits(64) offset = ExtendReg(m, extend_type, shift);
if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding  UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## PRFM (unscaled offset)

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see [Prefetch memory](#).

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	1	1	1	0	0	0	1	0	0	imm9									0	0	Rn			Rt									
size										opc																								

### Unscaled offset

```
PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.  
<type> is one of:

#### PLD

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

#### PLI

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

#### PST

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

#### L1

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

#### L2

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

#### L3

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

#### KEEP

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

#### STRM

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see [Prefetch memory](#).

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.  
This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## PSB CSYNC

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following DSB instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a NOP.

### System (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	1	1	1	1	1	1
																CRm				op2											

### System

PSB CSYNC

```

SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111'
    SEE "XPACLR1";
  when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 100'
    op = SystemHintOp_CSDB;
  when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0'
    op = otherwise SystemHintOp_BTI;
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction(); // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.
- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	1	1	1
CRm																opc															

## System

PSSBB

```

MemBarrierOp op;
MReqDomain domain;
MReqTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if otherwise HaveSBEExtUnallocatedEncoding() && CRm<3:0> == '0000' then
      op = ();
case CRm<3:2> of
  when '00' domain = MemBarrierOp_SB;
  else
    UNDEFINED;
case CRm<3:2> of
  when '00' domain = MReqDomain_OuterShareable;
  when '01' domain = MReqDomain_Nonshareable;
  when '10' domain = MReqDomain_InnerShareable;
  when '11' domain = MReqDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MReqTypes_Reads;
  when '10' types = MReqTypes_Writes;
  when '11' types = MReqTypes_All;
  otherwise
    if CRm<3:2> == '01' then
      if CRm<3:2> == '00' then
        op = MemBarrierOp_PSSBB;
      elseif CRm<3:2> == '00' && opc == '00' then
        op = MemBarrierOp_SSBB;
      elseif elseif CRm<3:2> == '01' then
        op = HaveSBEExtMemBarrierOp_PSSBB() && CRm<3:2> == '00' && opc == '11' then
          op = MemBarrierOp_SB;
    else
      types = MReqTypes_All;
      domain = MReqDomain_FullSystem;

```



Operation

```
case op of
  when MemBarrierOp_DSBDDataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier();
  when MemBarrierOp_SSBBSpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA();
  when (); MemBarrierOp_SBSpeculationBarrier();
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## RADDHN, RADDHN2

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see [ADDHN](#).

The RADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1				Rm			0	1	0	0	0	0										
U										o1																					

### Three registers, not all the same type

RADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

RAX1

Rotate and Exclusive OR rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.  
This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD  
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm				1	0	0	0	1	1	Rn				Rd						

Advanced SIMD

```
RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D
```

```
if !HaveSHA3Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt (Rd) ;
integer n = UInt (Rn) ;
integer m = UInt (Rm) ;
```

Assembler Symbols

- <Vd>Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn>Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled() ;

bits(128) Vm = V[m] ;
bits(128) Vn = V[n] ;
V[d] = Vn EOR (ROL (Vm<127:64>,1) :ROL (Vm<63:0>, 1)) ;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	1	0	1	1	0	0	1	0	1	1	1	1	1	0	0	0	0	0	0							0	0	0	0	0
Z							op							A							M	Rn							Rm			

## Integer

RET {<Xn>}

```
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UNDEFINED;
elseif pac && !if !pac && m != 0 then UnallocatedEncoding();
elseif pac && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = () then BranchType_INDIRUnallocatedEncoding();
    when '01' branch_type = ();

case op of
    when '00' branch_type = BranchType_INDCALLBranchType_JMP;
    when '10' branch_type = when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n != 31 then UnallocatedEncoding then
            if n != 31 then UNDEFINED;
        then
            n = 30;
            source_is_sp = TRUE;
```

## Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

## Operation

```
bits(64) target = X[n];
case branch_type of
  whenif pac then
    bits(64) modifier = if source_is_sp then BranchType_INDIR
    if InGuardedPage then
      if n == 16 || n == 17 then
        BTypeNext = '01';
      else
        BTypeNext = '11';
      else
        BTypeNext = '01';
    when BranchType_INDCALL
      BTypeNext = '10';
    when BranchType_RET
      BTypeNext = '00';
if pac then
  bits(64) modifier = if source_is_sp then SP[] else X[m];

  if use_key_a then
    target = AuthIA(target, modifier);
  else
    target = AuthIB(target, modifier);

if branch_type == BranchType_INDCALLBranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## RETAA, RETAB

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

Key A is used for RETAA, and key B is used for RETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated. The authenticated address is not written back to LR.

### Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	0	1	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1
Z							op					A				Rn					Rm										

### RETAA (M == 0)

RETAA

### RETAB (M == 1)

RETAB

```

integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UNDEFINED;
elseif pac && !if !pac && m != 0 then UnallocatedEncoding();
elseif pac && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = () then BranchType_INDIRUnallocatedEncoding;
    when '01' branch_type = ();

case op of
    when '00' branch_type = BranchType_INDCALLBranchType_JMP;
    when '10' branch_type = when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n != 31 then UnallocatedEncoding then
        if n != 31 then UNDEFINED;
()};

n = 30;
source_is_sp = TRUE;

```

Operation

```
bits(64) target = X[n];
case branch_type of
  whenif pac then
    bits(64) modifier = if source_is_sp then BranchType_INDIR
    if InGuardedPage then
      if n == 16 || n == 17 then
        BTypeNext = '01';
      else
        BTypeNext = '11';
      else
        BTypeNext = '01';
    when BranchType_INDCALL
      BTypeNext = '10';
    when BranchType_RET
      BTypeNext = '00';
if pac then
  bits(64) modifier = if source_is_sp then SP[] else X[m];

  if use_key_a then
    target = AuthIA(target, modifier);
  else
    target = AuthIB(target, modifier);

if branch_type == BranchType_INDCALLBranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

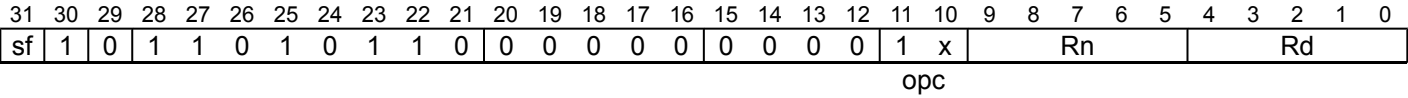
(old)	htmldiff from-	(new)
-------	----------------	-------



# REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction [REV64](#).



32-bit (sf == 0 && opc == 10)

```
REV <Wd>, <Wn>
```

64-bit (sf == 1 && opc == 11)

```
REV <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;
    if sf == '0' then UnallocatedEncoding();
    container_size = 64;
```

## Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index + 7:rev_index> = operand<index + 7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## REV16 (vector)

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	0	1	1	0	Rn				Rd					
U								o0																							

### Vector

REV16 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0),  H(1),  S(1),  D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx:  64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//   64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//   32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//   16+B = 2, 16+H = X, 16+S = X, 16+D = X
//   8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//   64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//   32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//   16+B = 1, 16+H = X, 16+S = X, 16+D = X
//   8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op)+UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize; (size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<I>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	Rn				Rd					
																						opc									

### 32-bit (sf == 0)

REV16 <Wd>, <Wn>

### 64-bit (sf == 1)

REV16 <Xd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64; if sf == '0' then UnallocatedEncoding();
    container_size = 64;
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index + 7:rev_index> = operand<index + 7:index>;
    index = index + 8;
    rev_index = rev_index - 8;

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## REV32 (vector)

Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	0	0	1	0	Rn				Rd					
U								o0																							

### Vector

REV32 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0),  H(1),  S(1),  D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx:  64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//   64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//   32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//   16+B = 2, 16+H = X, 16+S = X, 16+D = X
//   8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//   64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//   32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//   16+B = 1, 16+H = X, 16+S = X, 16+D = X
//   8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op)+UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize; (size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25-v29.05**, AdvSIMD **v27.01-v26.0**, pseudocode **v85-xml-00bet8\_rc3-v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## REV32

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	Rn				Rd					
sf											opc																				

### 64-bit

REV32 <Xd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;
    if sf == '0' then UnallocatedEncoding();
    container_size = 64;
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index + 7:rev_index> = operand<index + 7:index>;
    index = index + 8;
    rev_index = rev_index - 8;

X[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:20:16~~~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

**(old)**

**htmldiff from-**

**(new)**

## REV64

Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	0	0	1	0	Rn				Rd					
U								o0																							

### Vector

REV64 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0),  H(1),  S(1),  D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx:  64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//   64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//   32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//   16+B = 2, 16+H = X, 16+S = X, 16+D = X
//   8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//   64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//   32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//   16+B = 1, 16+H = X, 16+S = X, 16+D = X
//   8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op)+UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize; (size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

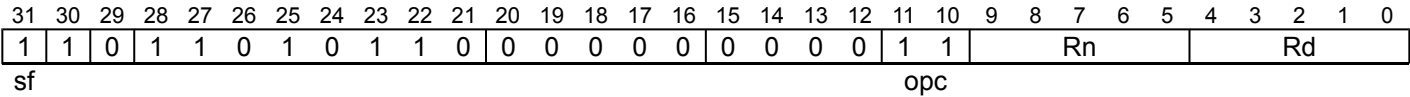
REV64

Reverse Bytes reverses the byte order in a 64-bit general-purpose register.

When assembling for ARMv8.2, an assembler must support this pseudo-instruction. It is OPTIONAL whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than ARMv8.2.

This is a pseudo-instruction of [REV](#). This means:

- The encodings in this description are named to match the encodings of [REV](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [REV](#) gives the operational pseudocode for this instruction.



64-bit

REV64 <Xd>, <Xn>

is equivalent to

[REV](#) <Xd>, <Xn>

Assembler Symbols

- <Xd>
- Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>
- Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of [REV](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

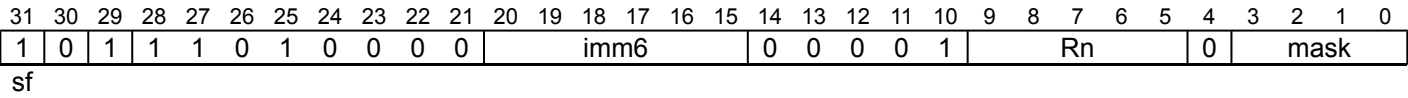
Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:20:18-06:16](#)~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RMIF

Performs a rotation right of a value held in a general purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

Integer  
(ARMv8.4)



Integer

```
RMIF <Xn>, #<shift>, #<mask>
```

```
if !HaveFlagManipulateExt() || sf != '1' then UNDEFINED;
integer lsb = () || sf != '1' then UnallocatedEncoding();
integer lsb = UInt(imm6);
integer n = UInt(Rn);
```

Assembler Symbols

- <Xn>

Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <shift>

Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
- <mask>

Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the "mask" field.

Operation

```
bits(4) tmp;
bits(64) tmpreg = X[n];
tmp = (tmpreg:tmpreg)<lsb+3:lsb>;
if mask<3> == '1' then PSTATE.N = tmp<3>;
if mask<2> == '1' then PSTATE.Z = tmp<2>;
if mask<1> == '1' then PSTATE.C = tmp<1>;
if mask<0> == '1' then PSTATE.V = tmp<0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## RSHRN, RSHRN2

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see [SHRN](#).

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	0	0	1	1	Rn				Rd					
immh										op																					

### Vector

```
RSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<shift>

Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt (immh:immb))
001x	(32-UInt (immh:immb))
01xx	(64-UInt (immh:immb))
1xxx	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25v29.05**, AdvSIMD **v27.01v26.0**, pseudocode **v85-xml-00bet8\_rc3v35.3**; Build timestamp: **2018-09-13T13:20:18-06:16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## RSUBHN, RSUBHN2

Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see [SUBHN](#).

The RSUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	Rm						0	1	1	0	0	0	Rn						Rd			
U										o1																					

### Three registers, not all the same type

RSUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

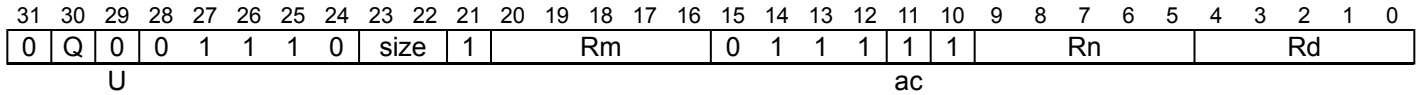
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SABA

Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

SABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SABAL, SABAL2

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABAL instruction extracts each source vector from the lower half of each source register, while the SABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	0	size	1	Rm						0	1	0	1	0	0	Rn						Rd					
U								op																									

### Three registers, not all the same type

```
SABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

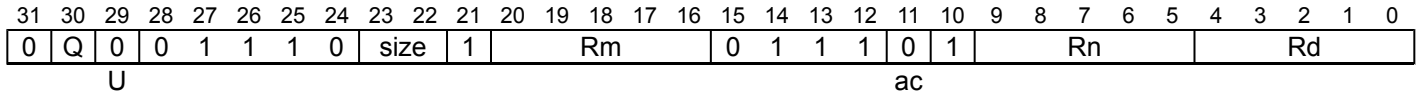
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SABD

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

SABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:18-06:16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABDL instruction writes the vector to the lower half of the destination register and clears the upper half, while the SABDL2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						0	1	1	1	0	0	Rn						Rd			
U										op																					

### Three registers, not all the same type

```
SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)    operand1 = Vpart[n, part];
bits(datasize)    operand2 = Vpart[m, part];
bits(2*datasize)  result;
integer element1;
integer element2;
bits(2*esize)  absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

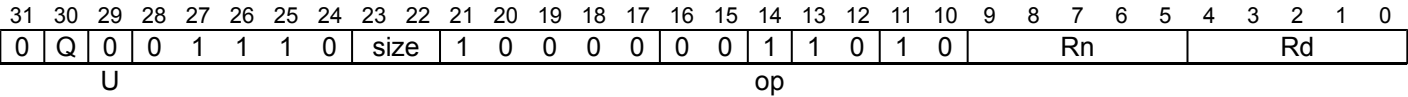
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# SADALP

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## Vector

```
SADALP <Vd>.<Ta>, <Vn>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

## Assembler Symbols

- <Vd>Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Ta>Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

- <Vn>Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <Tb>Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SADDL, SADDL2

Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SADDL instruction extracts each source vector from the lower half of each source register, while the SADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			0	0	0	0	0	0				Rn				Rd		
U								o1																							

### Three registers, not all the same type

SADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

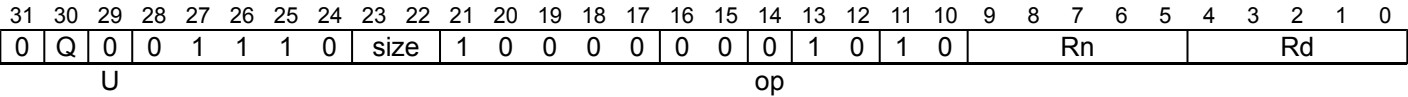
htmldiff from-

(new)

SADDLP

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Vector

```
SADDLP <Vd>.<Ta>, <Vn>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

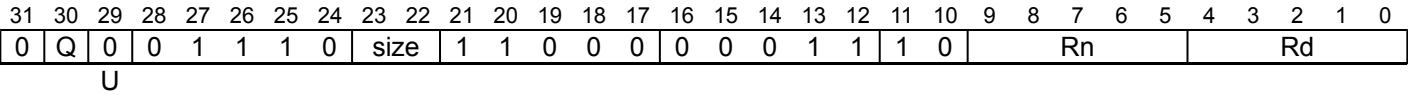
(old)	htmldiff from-	(new)
-------	----------------	-------



SADDLV

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Advanced SIMD

```
SADDLV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	H
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d] = sum<2*esize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:18+00:002018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SADDW, SADDW2

Signed Add Wide. This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The SADDW instruction extracts the second source vector from the lower half of the second source register, while the SADDW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						0	0	0	1	0	0	Rn						Rd			
U										o1																					

### Three registers, not all the same type

```
SADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SB

Speculation Barrier is a barrier that controls speculation.

The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	(0)	(0)	(0)	(0)	1	1	1	1	1	1	1	1
																CRm				opc											

## System

SB

```

MemBarrierOp op;
MBReqDomain domain;
MBReqTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if HaveSBExt() && CRm<3:0> == '0000' then
      op = MemBarrierOp_SB;
    else
      UNDEFINED;

case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;
  otherwise
    if CRm<3:2> == '01' then
      op = MemBarrierOp_PSSBB;
    elsif CRm<3:2> == '00' && opc == '00' then
      op = MemBarrierOp_SSBB;
    elsif HaveSBExt() && CRm<3:2> == '00' && opc == '11' then
      op = MemBarrierOp_SB;
    else
      types = MBReqTypes_All;
      domain = MBReqDomain_FullSystem;

```

Operation

```
case op of
  when MemBarrierOp\_DSBDataSynchronizationBarrier(domain, types);
  when MemBarrierOp\_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp\_ISBInstructionSynchronizationBarrier();
  when MemBarrierOp\_SSBBSpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp\_PSSBBSpeculativeSynchronizationBarrierToPA();
  when MemBarrierOp\_SBSpeculationBarrier();
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file	htmldiff from-	(new)
-------------	----------------	-------

## SBFM

Signed Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If  $\langle \text{imms} \rangle$  is greater than or equal to  $\langle \text{immr} \rangle$ , this copies a bitfield of  $(\langle \text{imms} \rangle - \langle \text{immr} \rangle + 1)$  bits starting from bit position  $\langle \text{immr} \rangle$  in the source register to the least significant bits of the destination register.

If  $\langle \text{imms} \rangle$  is less than  $\langle \text{immr} \rangle$ , this copies a bitfield of  $(\langle \text{imms} \rangle + 1)$  bits from the least significant bits of the source register to bit position  $(\text{regsize} - \langle \text{immr} \rangle)$  of the destination register, where  $\text{regsize}$  is the destination register size of 32 or 64 bits.

In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases [ASR \(immediate\)](#), [SBFIZ](#), [SBFX](#), [SXTB](#), [SXTH](#), and [SXTW](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	1	1	0	N	immr						imms						Rn				Rd					
opc																															

### 32-bit (sf == 0 && N == 0)

SBFM  $\langle \text{Wd} \rangle$ ,  $\langle \text{Wn} \rangle$ ,  $\# \langle \text{immr} \rangle$ ,  $\# \langle \text{imms} \rangle$

### 64-bit (sf == 1 && N == 1)

SBFM  $\langle \text{Xd} \rangle$ ,  $\langle \text{Xn} \rangle$ ,  $\# \langle \text{immr} \rangle$ ,  $\# \langle \text{imms} \rangle$

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

case opc of
  when '00' inzero = TRUE;  extend = TRUE;    // SBFM
  when '01' inzero = FALSE; extend = FALSE;   // BFM
  when '10' inzero = TRUE;  extend = FALSE;   // UBFM
  when '11' UNDEFINED;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

R = when '11' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

## Assembler Symbols

$\langle \text{Wd} \rangle$	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
$\langle \text{Wn} \rangle$	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
$\langle \text{Xd} \rangle$	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
$\langle \text{Xn} \rangle$	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.
	For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
<imms>	For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.
	For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Of variant	Is preferred when
<a href="#">ASR (immediate)</a>	32-bit	<code>imms == '011111'</code>
<a href="#">ASR (immediate)</a>	64-bit	<code>imms == '111111'</code>
<a href="#">SBFIZ</a>		<code>UInt(imms) &lt; UInt(immr)</code>
<a href="#">SBFX</a>		<code>BFXPreferred(sf, opc&lt;1&gt;, imms, immr)</code>
<a href="#">SXTB</a>		<code>immr == '000000' &amp;&amp; imms == '000111'</code>
<a href="#">SXTH</a>		<code>immr == '000000' &amp;&amp; imms == '001111'</code>
<a href="#">SXTW</a>		<code>immr == '000000' &amp;&amp; imms == '011111'</code>

Operation

```

bits(datasize) dst = if inzero then Zeros\(\) else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);

```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13](#)~~2018-06-16T09~~:[0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	<a href="#">htmldiff</a>	from-	(new)
-------	--------------------------	-------	-------



## SCVTF (vector, fixed-point)

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn					Rd								
U									immh																										

### Scalar

SCVTF <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - (()) then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn				Rd					
U									immb																						

Vector

SCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - (()) then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SCVTF (vector, integer)

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

### Scalar half precision

SCVTF <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

### Scalar single-precision and double-precision

SCVTF <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

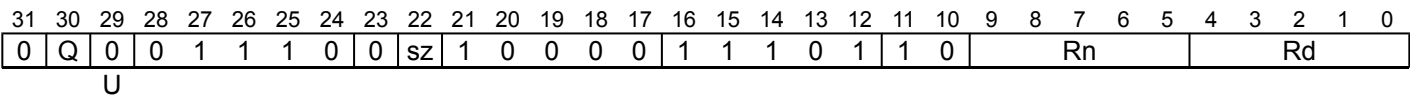
Vector half precision

```
SCVTF <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
SCVTF <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the vector half precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
FPRounding rounding = FPRoundingMode(FPCR);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

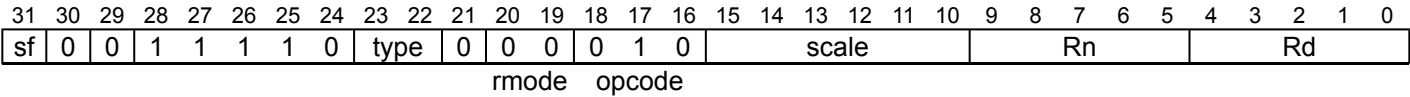
(old)	htmldiff from-	(new)
-------	----------------	-------

SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.



**32-bit to half-precision (sf == 0 && type == 11)**  
(ARMv8.2)

SCVTF <Hd>, <Wn>, #<fbits>

**32-bit to single-precision (sf == 0 && type == 00)**

SCVTF <Sd>, <Wn>, #<fbits>

**32-bit to double-precision (sf == 0 && type == 01)**

SCVTF <Dd>, <Wn>, #<fbits>

**64-bit to half-precision (sf == 1 && type == 11)**  
(ARMv8.2)

SCVTF <Hd>, <Xn>, #<fbits>

**64-bit to single-precision (sf == 1 && type == 00)**

SCVTF <Sd>, <Xn>, #<fbits>

**64-bit to double-precision (sf == 1 && type == 01)**

SCVTF <Dd>, <Xn>, #<fbits>



```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if when '10' UnallocatedEncoding();
        when '11'
            if HaveFP16Ext() then
                fltsize = 16;
            else
                UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - else UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
    when '00 11' // FCVTZ
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00' // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    otherwise
        UnallocatedEncoding;
    otherwise
        UNDEFINED; ();

```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	<p>For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".</p> <p>For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".</p>

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
    V[d] = fltval;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

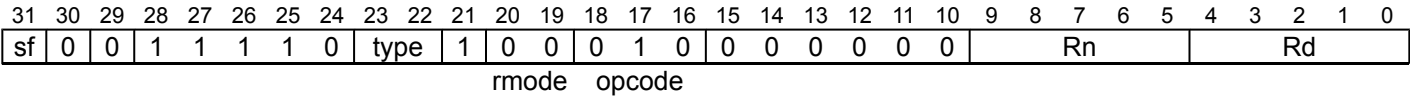
(old)	htmldiff from-	(new)
-------	----------------	-------

SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**32-bit to half-precision (sf == 0 && type == 11)**  
(ARMv8.2)

SCVTF <Hd>, <Wn>

**32-bit to single-precision (sf == 0 && type == 00)**

SCVTF <Sd>, <Wn>

**32-bit to double-precision (sf == 0 && type == 01)**

SCVTF <Dd>, <Wn>

**64-bit to half-precision (sf == 1 && type == 11)**  
(ARMv8.2)

SCVTF <Hd>, <Xn>

**64-bit to single-precision (sf == 1 && type == 00)**

SCVTF <Sd>, <Xn>

**64-bit to double-precision (sf == 1 && type == 01)**

SCVTF <Dd>, <Xn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64 ();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp\_CVT\_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp\_CVT\_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
    V[d] = fltval;
  when FPConvOp\_MOV\_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp\_MOV\_ItoF
    intval = X[n];
    fltval = intval<fltsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp\_CVT\_FtoI\_JS
    fltval = V[n];
    intval = FPToFixedJS(fltval, FPCR, TRUE);
    X[d] = ZeroExtend(intval<31:0>, 64);

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:04Z2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## SDOT (by element)

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

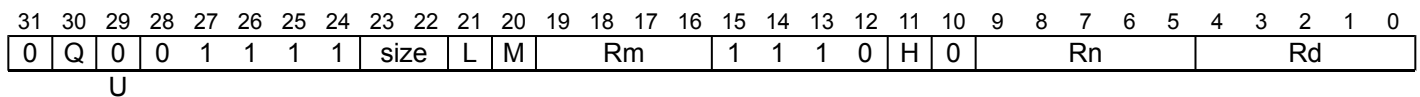
Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

[ID\\_AA64ISAR0\\_EL1](#).DP indicates whether this instruction is supported.

### Vector

(ARMv8.2)



### Vector

SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.[4B](#)[<index>]

```

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U=='0');

integer d = if size != '10' then ReservedValue();
boolean signed = (U=='0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the element index, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## SDOT (vector)

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

[ID\\_AA64ISAR0\\_EL1](#).DP indicates whether this instruction is supported.

### Three registers of the same type (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	0				Rm			1	0	0	1	0	1				Rn					Rd	
U																															

### Three registers of the same type

SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```

if !HaveDOTPExt() then UNDEFINED;
if size!= '10' then UNDEFINED;
boolean signed = (U=='0');
integer d =if size!= '10' then ReservedValue();
boolean signed = (U=='0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = Sint(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = Sint(Elem[operand2, 4 * e + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

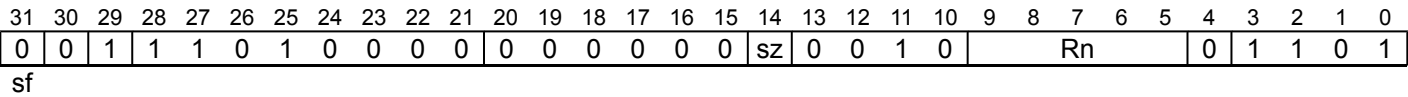
(old)	htmldiff from-	(new)
-------	----------------	-------

## SETF8, SETF16

Set the PSTATE.NZV flags based on the value in the specified general-purpose register. SETF8 treats the value as an 8 bit value, and SETF16 treats the value as an 16 bit value.

The PSTATE.C flag is not affected by these instructions.

### Integer (ARMv8.4)



#### SETF8 (sz == 0)

SETF8 <Wn>

#### SETF16 (sz == 1)

SETF16 <Wn>

```
if !HaveFlagManipulateExt() || sf != '0' then UNDEFINED;
integer msb = if sz=='1' then 15 else 7;
integer n = () || sf != '0' then UnallocatedEncoding();
integer msb = if sz=='1' then 15 else 7;
integer n = UInt (Rn) ;
```

### Assembler Symbols

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

### Operation

```
bits(32) tmpreg = X[n];
PSTATE.N = tmpreg<msb>;
PSTATE.Z = if (tmpreg<msb:0> == Zeros(msb+1)) then '1' else '0';
PSTATE.V = tmpreg<msb+1> EOR tmpreg<msb>;
//PSTATE.C unchanged;
```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see [Wait for Event mechanism and Send event](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	1	1	1
																CRm								op2							

## System

SEV

```

SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111'
    SEE "XPACLRI";
  when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction();           // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction();   // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 100'
    op = SystemHintOp_CSDB;
  when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0'
    op = otherwise SystemHintOp_BTI;
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction();           // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a WFE instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	1	1	1	1	1
																CRm				op2											

## System

SEVL

[SystemHintOp](#) op;

case CRm:op2 of

```
when '0000 000' op = SystemHintOp NOP;
when '0000 001' op = SystemHintOp YIELD;
when '0000 010' op = SystemHintOp WFE;
when '0000 011' op = SystemHintOp WFI;
when '0000 100' op = SystemHintOp SEV;
when '0000 101' op = SystemHintOp SEVL;
when '0000 111'
    SEE "XPACLRI";
```

```
when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
```

```
when '0010 000'
    if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp ESB;
```

```
when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp PSB;
```

```
when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp TSB;
```

```
when '0010 100'
    op = SystemHintOp CSDB;
```

```
when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
```

```
when '0100 xx0'
    op = otherwise SystemHintOp BTI;
    BTypeCompatible = BTypeCompatible BTI(op2<2:1>);
```

```
otherwise EndOfInstruction(); // Instruction executes as NOP
```

## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)bet8\_rc3~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

SHA1C

SHA1 hash update (choose).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm				0	0	0	0	0	0	Rn				Rd						

Advanced SIMD

SHA1C <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;() thenUnallocatedEncoding();
```

Assembler Symbols

- <Qd>

Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Sn>

Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm>

Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32) Y = V[n]; // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
    t = SHAchoose(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y : X, 32);
V[d] = X;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SHA1H

SHA1 fixed rotate.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	Rn				Rd					

Advanced SIMD

SHA1H <Sd>, <Sn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED; (then UnallocatedEncoding();)
```

Assembler Symbols

- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(32) operand = V[n]; // read element [0] only, [1-3] zeroed
V[d] = ROL(operand, 30);
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SHA1M

SHA1 hash update (majority).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0			Rm			0	0	1	0	0	0			Rn					Rd		

### Advanced SIMD

SHA1M <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED; () then UnallocatedEncoding();
```

### Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled\(\);

bits(128) X = V[d];
bits(32) Y = V[n]; // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
    t = SHAmajority(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y : X, 32);
V[d] = X;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)~~bet8\_rc3~~[v35.3](#); Build timestamp: [2018-09-13T13:20:09](#)~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1P

SHA1 hash update (parity).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm				0	0	0	1	0	0	Rn				Rd						

Advanced SIMD

SHA1P <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED; () then UnallocatedEncoding();
```

Assembler Symbols

- <Qd>

Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Sn>

Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm>

Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled\(\);

bits(128) X = V[d];
bits(32) Y = V[n]; // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
    t = SHAparity(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y : X, 32);
V[d] = X;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)~~bet8\_rc3~~[v35.3](#); Build timestamp: [2018-09-13T13:20:09](#)~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1SU0

SHA1 schedule update 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm				0	0	1	1	0	0	Rn				Rd						

Advanced SIMD

SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED; ({} then UnallocatedEncoding();)
```

Assembler Symbols

- <Vd>

Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn>

Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm>

Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled\(\);

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;

result = operand2<63:0> : operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa **v30.25v29.05**, AdvSIMD **v27.01v26.0**, pseudocode **v85-xml-00bet8\_rc3v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1SU1

SHA1 schedule update 1.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	Rn				Rd					

Advanced SIMD

SHA1SU1 <Vd>.4S, <Vn>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED; ({} then UnallocatedEncoding();)
```

Assembler Symbols

- <Vd>Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn>Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand1 EOR LSR(operand2, 32);
result<31:0> = ROL(T<31:0>, 1);
result<63:32> = ROL(T<63:32>, 1);
result<95:64> = ROL(T<95:64>, 1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d] = result;
```

Operational information

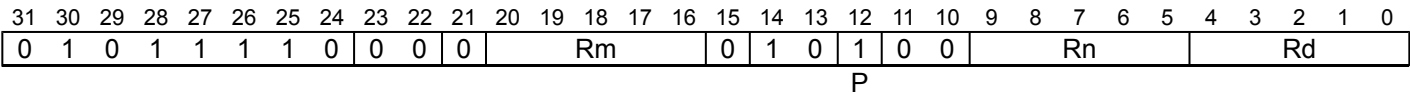
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256H2

SHA256 hash update (part 2).



Advanced SIMD

SHA256H2 <Qd>, <Qn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
boolean part1 = (P == '0'); if part1 then UnallocatedEncoding();
boolean part1 = (P == '0');
```

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
if part1 then
    result = SHA256hash(V[d], V[n], V[m], TRUE);
else
    result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;
```

Operational information

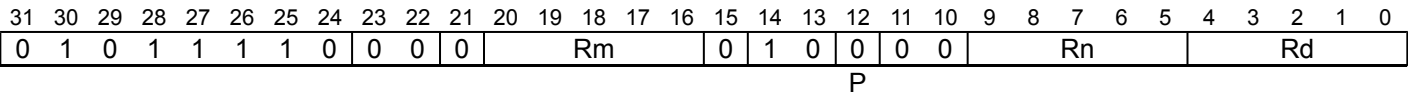
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256H

SHA256 hash update (part 1).



Advanced SIMD

SHA256H <Qd>, <Qn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
boolean part1 = (P == '0'); if part1 then UnallocatedEncoding();
boolean part1 = (P == '0');
```

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
if part1 then
    result = SHA256hash(V[d], V[n], V[m], TRUE);
else
    result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256SU0

SHA256 schedule update 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	Rn				Rd					

Advanced SIMD

SHA256SU0 <Vd>.4S, <Vn>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA256Ext() then UNDEFINED;() thenUnallocatedEncoding();
```

Assembler Symbols

- <Vd>Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn>Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand2<31:0> : operand1<127:32>;
bits(32) elt;

for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## SHA256SU1

SHA256 schedule update 1.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0			Rm			0	1	1	0	0	0			Rn					Rd		

### Advanced SIMD

SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED; () then UnallocatedEncoding();
```

### Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled\(\);

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
bits(128) T0 = operand3<31:0> : operand2<127:32>;
bits(64) T1;
bits(32) elt;

T1 = operand3<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e - 2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

(old)

htmldiff from-

(new)

## SHA512H2

SHA512 Hash update part 2 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when [ARMv8.2-SHA](#) is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm				1	0	0	0	0	1	Rn				Rd						

### Advanced SIMD

SHA512H2 <Qd>, <Qn>, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

### Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) NSigma0;
bits(64) tmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

NSigma0 = ROR(Y<63:0>, 28) EOR ROR(Y<63:0>, 34) EOR ROR(Y<63:0>, 39);
Vtmp<127:64> = (X<63:0> AND Y<127:64>) EOR (X<63:0> AND Y<63:0>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 + W<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR ROR(Vtmp<127:64>, 39);
Vtmp<63:0> = (Vtmp<127:64> AND Y<63:0>) EOR (Vtmp<127:64> AND Y<127:64>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + NSigma0 + W<63:0>);

V[d] = Vtmp;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

(old)

htmldiff from-

(new)

## SHA512H

SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register. This instruction is implemented only when [ARMv8.2-SHA](#) is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm				1	0	0	0	0	0	Rn				Rd						

### Advanced SIMD

SHA512H <Qd>, <Qn>, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

### Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) MSigma1;
bits(64) tmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

MSigma1 = ROR(Y<127:64>, 14) EOR ROR(Y<127:64>, 18) EOR ROR(Y<127:64>, 41);
Vtmp<127:64> = (Y<127:64> AND X<63:0>) EOR (NOT(Y<127:64>) AND X<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + W<127:64>);
tmp = Vtmp<127:64> + Y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR ROR(tmp, 41);
Vtmp<63:0> = (tmp AND Y<127:64>) EOR (NOT(tmp) AND X<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + W<63:0>);
V[d] = Vtmp;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

(old)

htmldiff from-

(new)

# SHA512SU0

SHA512 Schedule Update 0 takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SHA* is implemented.

## Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0										
												Rn				Rd															

## Advanced SIMD

SHA512SU0 <Vd>.*2D*, <Vn>.*2D*

```
if !HaveSHA512Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt (Rd);
integer n = UInt (Rn);
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig0;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) W = V[d];
sig0 = ROR(W<127:64>, 1) EOR ROR(W<127:64>, 8) EOR ('0000000':W<127:71>);
(W<127:64>, 8) EOR '0000000':W<127:71>;
Vtmp<63:0> = W<63:0> + sig0;
sig0 = ROR(X<63:0>, 1) EOR ROR(X<63:0>, 8) EOR ('0000000':X<63:7>);
(X<63:0>, 8) EOR '0000000':X<63:7>;
Vtmp<127:64> = W<127:64> + sig0;
V[d] = Vtmp;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SHA512SU1

SHA512 Schedule Update 1 takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when [ARMv8.2-SHA](#) is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1					Rm		1	0	0	0	1	0									Rd

### Advanced SIMD

SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt (Rd);
integer n = UInt (Rn);
integer m = UInt (Rm);
```

### Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig1;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

sig1 = ROR(X<127:64>, 19) EOR ROR(X<127:64>, 61) EOR ('000000':X<127:70>);
(X<127:64>, 61) EOR '000000':X<127:70>;
Vtmp<127:64> = W<127:64> + sig1 + Y<127:64>;
sig1 = ROR(X<63:0>, 19) EOR ROR(X<63:0>, 61) EOR ('000000':X<63:6>);
(X<63:0>, 61) EOR '000000':X<63:6>;
Vtmp<63:0> = W<63:0> + sig1 + Y<63:0>;
V[d] = Vtmp;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



(old)

htmldiff from-

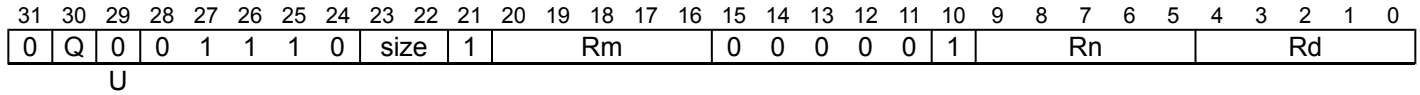
(new)

## SHADD

Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see [SRHADD](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

SHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SHL

Shift Left (immediate). This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	1	0	1	0	1	1	Rn					Rd			
immh																															

### Scalar

SHL <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	1	0	1	0	1	Rn			Rd						
immh																															

### Vector

SHL <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

## Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in "immh:immb":

immh	<shift>
0xxx	RESERVED
1xxx	(UInt (immh:immb) - 64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt (immh:immb) - 8)
001x	(UInt (immh:immb) - 16)
01xx	(UInt (immh:immb) - 32)
1xxx	(UInt (immh:immb) - 64)

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = LSL(Elem[operand, e, esize], shift);

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa [v30.25v29.05](#), AdvSIMD [v27.01v26.0](#), pseudocode [v85-xml-00bet8\\_rc3v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SHLL, SHLL2

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SHLL instruction extracts vector elements from the lower half of the source register, while the SHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	1	0	0	1	1	1	0	Rn				Rd					

### Vector

```
SHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue{};
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = esize;
boolean unsigned = FALSE; // Or TRUE without change of functionality
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<shift> Is the left shift amount, which must be equal to the source element width in bits, encoded in “size”:

size	<shift>
00	8
01	16
10	32
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25-v29.05**, AdvSIMD **v27.01-v26.0**, pseudocode **v85-xml-00bet8\_rc3-v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SHRN, SHRN2

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see [RSHRN](#).

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	0	0	0	1	Rn				Rd					
immh										op																					

### Vector

SHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED



<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immh”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immh))
001x	(32-UInt(immh:immh))
01xx	(64-UInt(immh:immh))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SHSUB

Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						0	0	1	0	0	1	Rn						Rd			
U																															

### Three registers of the same type

```
SHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

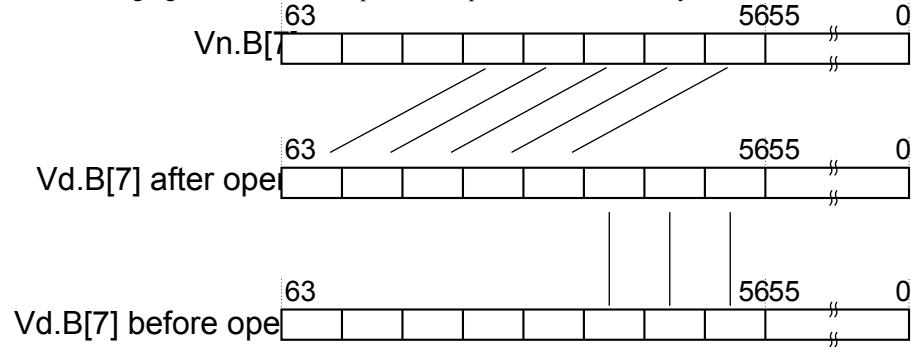
Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SLI

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost. The following figure shows an example of the operation of shift left by 3 for an 8-bit vector element.



Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000			immb			0	1	0	1	0	1	1	Rn					Rd				
immh																															

### Scalar

SLI <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	0	1	0	1	Rn					Rd				
immh																															

Vector

```
SLI <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(UInt(immh:immb) - 64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb) - 8)
001x	(UInt(immh:immb) - 16)
01xx	(UInt(immh:immb) - 32)
1xxx	(UInt(immh:immb) - 64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
bits(esize) mask = LSL(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSL(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SM3PARTW1

SM3PARTW1 takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when [ARMv8.2-SM](#) is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1					Rm		1	1	0	0	0	0					Rn				Rd

### Advanced SIMD

SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM3Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

### Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;

result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>,15):ROL(Vm<95:64>,15):ROL(Vm<63:32>,15));

for i = 0 to 3
  if i == 3 then
    result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>,15));
  result<(32*i)+31:(32*i)> = result<(32*i)+31:(32*i)> EOR ROL(result<(32*i)+31:(32*i)>,15) EOR ROL(re
  V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa [v30.25-v29.05](#), AdvSIMD [v27.01-v26.0](#), pseudocode [v85-xml-00bet8\\_rc3-v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when [ARMv8.2-SM](#) is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1					Rm		1	1	0	0	0	1									Rd

### Advanced SIMD

SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM3Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

### Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;
bits(128) tmp;
bits(32) tmp2;
tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7) : ROL(Vm<95:64>, 7) : ROL(Vm<63:32>, 7) : ROL(Vm<31:0>, 7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR ROL(tmp2, 23);
result<127:96> = result<127:96> EOR tmp2;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa [v30.25-v29.05](#), AdvSIMD [v27.01-v26.0](#), pseudocode [v85-xml-00bet8\\_rc3-v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

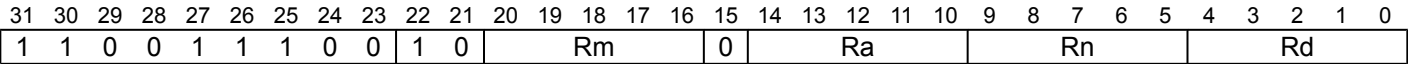
(new)

SM3SS1

SM3SS1 rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD  
(ARMv8.2)



Advanced SIMD

SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S

```
if !HaveSM3Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt (Rd) ;
integer n = UInt (Rn) ;
integer m = UInt (Rm) ;

integer a = UInt (Ra) ;
```

Assembler Symbols

- <Vd>
- Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn>
- Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>
- Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Va>
- Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) Va = V[a];
Vd<127:96> = ROL( (ROL(Vn<127:96>,12) + Vm<127:96> + Va<127:96>) , 7);
Vd<95:0> = Zeros();
V[d] = Vd;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

(old)

htmldiff from-

(new)

## SM3TT1A

SM3TT1A takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm				1	0	imm2	0	0	Rn				Rd							

### Advanced SIMD

```
SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

```
if !HaveSM3Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

### Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm,i,32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>,12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1 + Vd<31:0> + SS2 + WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>,9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SM3TT1B

SM3TT1B takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm				1	0	imm2	0	1	Rn				Rd							

### Advanced SIMD

```
SM3TT1B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

```
if !HaveSM3Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

### Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm,i,32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>,12);
TT1 = (Vd<127:96> AND Vd<63:32>) OR (Vd<127:96> AND Vd<95:64>) OR (Vd<63:32> AND Vd<95:64>);
TT1 = (TT1 + Vd<31:0> + SS2 + WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>,9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## SM3TT2A

SM3TT2A takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	1	1	1	0	0	1	0	Rm				1	0	imm2		1	0	Rn				Rd							

### Advanced SIMD

```
SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

```
if !HaveSM3Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

### Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT1;

Wj = Elem[Vm,i,32];
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2 + Vd<31:0> + Vn<127:96> + Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>,19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2,9) EOR ROL(TT2,17);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SM3TT2B

SM3TT2B takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	1	1	1	0	0	1	0	Rm				1	0	imm2		1	1	Rn				Rd							

### Advanced SIMD

```
SM3TT2B <Vd>.S, <Vn>.S, <Vm>.S[<imm2>]
```

```
if !HaveSM3Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

### Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm,i,32];
TT2 = (Vd<127:96> AND Vd<95:64>) OR (NOT(Vd<127:96>) AND Vd<63:32>);
TT2 = (TT2 + Vd<31:0> + Vn<127:96> + Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>,19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2,9) EOR ROL(TT2,17);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SM4E

SM4 Encode takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when [ARMv8.2-SM](#) is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	1		Rn				Rd				

### Advanced SIMD

SM4E <Vd>.4S, <Vn>.4S

```
if !HaveSM4Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled\(\);

bits(128) Vn = V[n];
bits(32) intval;
bits(8) sboxout;
bits(128) roundresult;
bits(32) roundkey;
integer index;

roundresult = V[d];
for index = 0 to 3
    roundkey = Elem[Vn, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
V[d] = roundresult;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

<b>(old)</b>	<b>htmldiff from-</b>	<b>(new)</b>
--------------	-----------------------	--------------

## SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when [ARMv8.2-SM](#) is implemented.

### Advanced SIMD (ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm				1	1	0	0	1	0	Rn				Rd						

### Advanced SIMD

SM4EKEY <Vd>.<sub>4S</sub>, <Vn>.<sub>4S</sub>, <Vm>.<sub>4S</sub>

```
if !HaveSM4Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

### Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(32) intval;
bits(8) sboxout;
bits(128) result;
bits(32) const;
bits(128) roundresult;
integer index;

roundresult = V[n];
for index = 0 to 3
    const = Elem[Vm, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
V[d] = roundresult;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

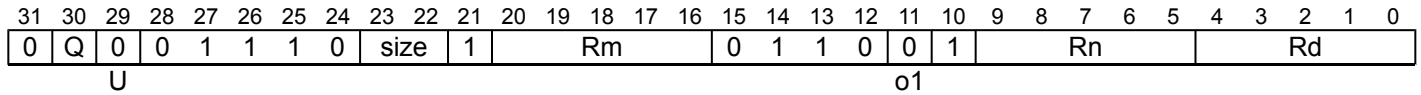
(old)	htmldiff from-	(new)
-------	----------------	-------



## SMAX

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

SMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

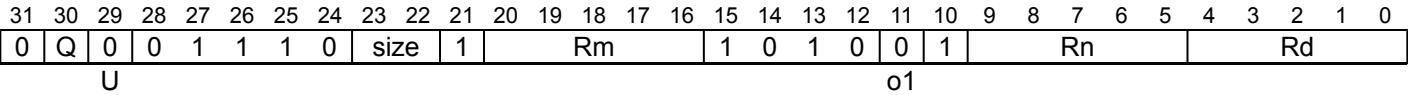
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SMAXP

Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

```
SMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

### Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

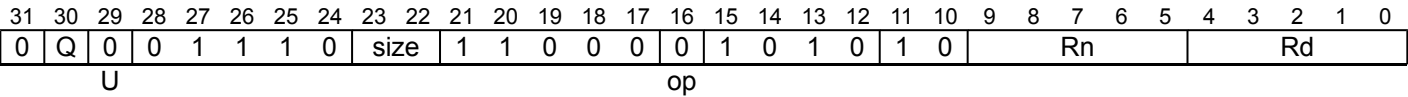
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

SMAXV

Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Advanced SIMD

```
SMAXV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

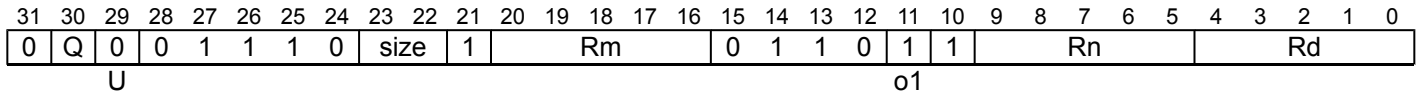
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SMIN

Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

SMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

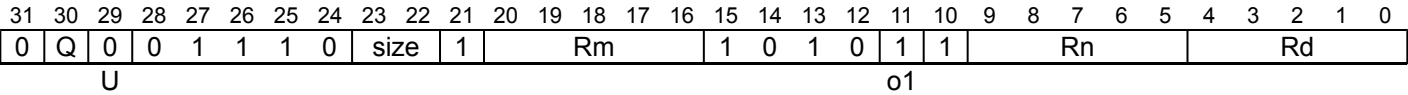
(old)	htmldiff from-	(new)
-------	----------------	-------



# SMINP

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## Three registers of the same type

```
SMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

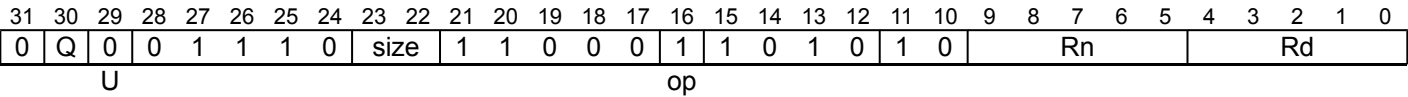
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

SMINV

Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Advanced SIMD

```
SMINV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SMLAL, SMLAL2 (by element)

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The SMLAL instruction extracts vector elements from the lower half of the first source register, while the SMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_ELI](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M			Rm		0	0	1	0	H	0											
U										o2																					

### Vector

SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```

integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UNDEFINED;

integer d = otherwise UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2) < 2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25v29.05**, AdvSIMD **v27.01v26.0**, pseudocode **v85-xml-00bet8\_rc3v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLAL instruction extracts each source vector from the lower half of each source register, while the SMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	0	0	1	1	1	0	size	1	Rm						1	0	0	0	0	0	Rn						Rd							
U										o1																									

### Three registers, not all the same type

SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25-v29.05**, AdvSIMD **v27.01-v26.0**, pseudocode **v85-xml-00bet8\_rc3-v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## SMLSL, SMLSL2 (by element)

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts vector elements from the lower half of the first source register, while the SMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M			Rm			0	1	1	0	H	0										
U										o2																					

### Vector

SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```

integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UNDEFINED;

integer d = otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(idxdsize)  operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2) < 2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25v29.05**, AdvSIMD **v27.01v26.0**, pseudocode **v85-xml-00bet8\_rc3v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts each source vector from the lower half of each source register, while the SMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						1	0	1	0	0	0	Rn				Rd					
U										o1																					

### Three registers, not all the same type

```
SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25-v29.05**, AdvSIMD **v27.01-v26.0**, pseudocode **v85-xml-00bet8\_rc3-v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SMOV

Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	imm5					0	0	1	0	1	1	Rn					Rd				

### 32-bit (Q == 0)

SMOV <Wd>, <Vn>.<Ts>[<index>]

### 64-reg,SMOV-64-reg (Q == 1)

SMOV <Xd>, <Vn>.<Ts>[<index>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when 'xxxxx1' size = 0;      // SMOV [WX]d, Vn.B
    when 'xxxx10' size = 1;      // SMOV [WX]d, Vn.H
    when 'lxx100' size = 2;      // SMOV Xd, Vn.S
    otherwise UNDEFINED;

integer idxsize = if imm5<4> == '1' then 128 else 64;
integer index = otherwise UnallocatedEncoding();

integer idxsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4>:size+1);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

## Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in "imm5":

imm5	<Ts>
xxx00	RESERVED
xxxx1	B
xxx10	H

For the 64-reg,SMOV-64-reg variant: is an element size specifier, encoded in "imm5":

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

<index> For the 32-bit variant: is the element index encoded in "imm5":

imm5	<index>
xxx00	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>

imm5	<index>
xx000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>

```

CheckFPAdvSIMDEnabled64();
bits(idxsdsz) operand = V[n];

X[d] = SignExtend(Elem[operand, index, esize], datasize);

```

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)

## SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts vector elements from the lower half of the first source register, while the SMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M			Rm			1	0	1	0	H	0				Rn					Rd	
U																															

### Vector

```
SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UNDEFINED;

integer d = otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)    operand1 = Vpart[n, part];
bits(idxdsize)    operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:20:18-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## SMULL, SMULL2 (vector)

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts each source vector from the lower half of each source register, while the SMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	Q	0	0	1	1	1	0	size	1	Rm						1	1	0	0	0	0	Rn						Rd				
U																																

### Three registers, not all the same type

```
SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)    operand1 = Vpart[n, part];
bits(datasize)    operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1 * element2)<2*esize-1:0>;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

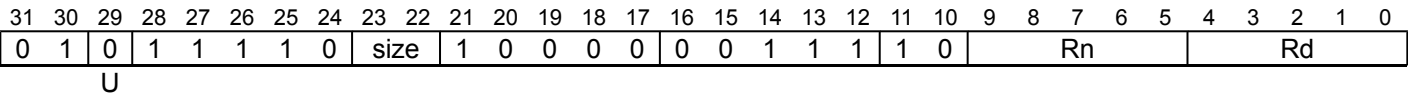
(new)

SQABS

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set. Depending on the settings in the *CPACR\_ELI*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



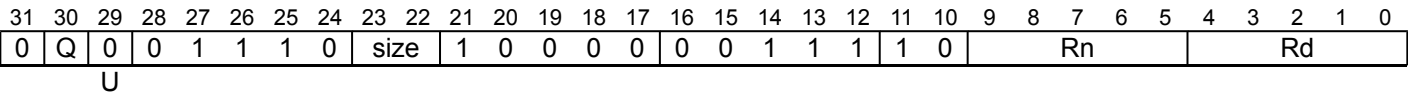
Scalar

```
SQABS <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector



Vector

```
SQABS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>** Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
------	--

## Operation

```

checkFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

(new)

# SQADD

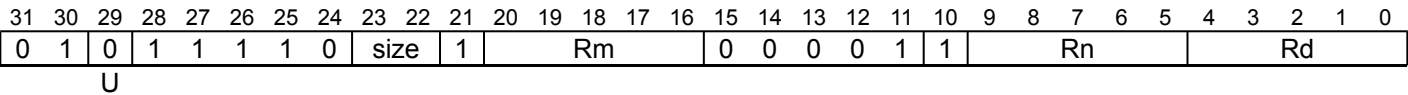
Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit `FPSR.QC` is set.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

## Scalar

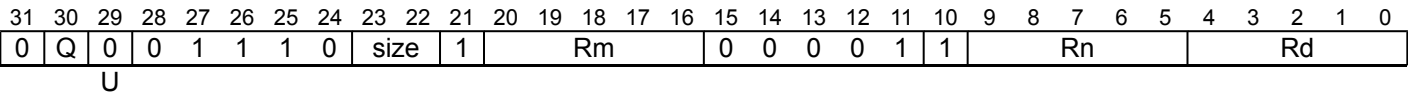


## Scalar

```
SQADD <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

## Vector



## Vector

```
SQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

## Assembler Symbols

- <V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D
- <d>

Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<b>&lt;Vm&gt;</b>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
-------------------	---

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## SQDMLAL, SQDMLAL2 (by element)

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The SQDMLAL instruction extracts vector elements from the lower half of the first source register, while the SQDMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size	L	M			Rm			0	0	1	1	H	0								Rd		

o2

### Scalar

SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

```

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);    Rmhi = M;
    otherwise UNDEFINED;
integer d = otherwise UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M			Rm			0	0	1	1	H	0								Rd		

o2

Vector

SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);    Rmhi = M;
    otherwise UNDEFINED;

integer d = otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:



size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)    operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

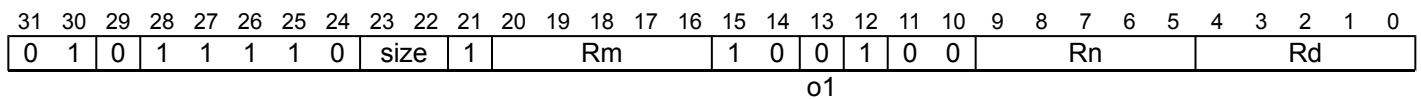
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The SQDMLAL instruction extracts each source vector from the lower half of each source register, while the SQDMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar



### Scalar

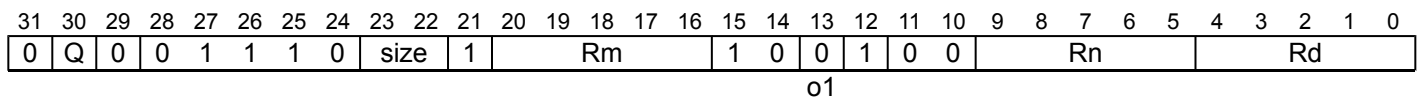
SQDMLAL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

### Vector



### Vector

SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

## Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQDMLSL, SQDMLSL2 (by element)

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The SQDMLSL instruction extracts vector elements from the lower half of the first source register, while the SQDMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	1	0	1	1	1	1	1	size	L	M			Rm			0	1	1	1	H	0													
																	o2										Rn				Rd			

### Scalar

SQDMLSL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

```

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UNDEFINED;
integer d = otherwise UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	Q	0	0	1	1	1	1	size	L	M			Rm			0	1	1	1	H	0													
																	o2										Rn				Rd			

Vector

SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);    Rmhi = M;
    otherwise UNDEFINED;

integer d = otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQDMLSL, SQDMLSL2 (vector)

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

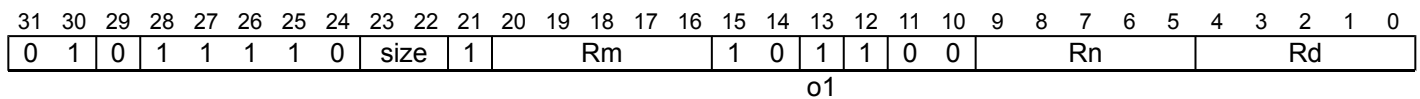
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The SQDMLSL instruction extracts each source vector from the lower half of each source register, while the SQDMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar



### Scalar

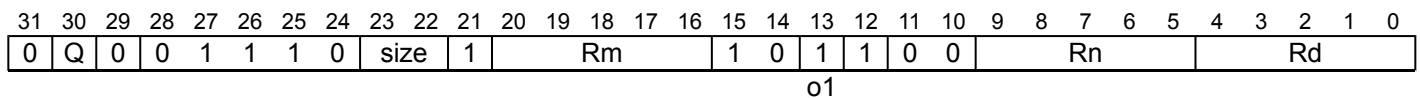
SQDMLSL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

### Vector



### Vector

SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```



## Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQDMULH (by element)

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see [SQRDMULH](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size	L	M				Rm		1	1	0	0	H	0				Rn				Rd		
op																															

### Scalar

SQDMULH [<V>](#)[<d>](#), [<V>](#)[<n>](#), [<Vm>](#).[<Ts>](#)[[<index>](#)]

```

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
integer d = otherwise UnallocatedEncoding\(\);
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M				Rm		1	1	0	0	H	0				Rn				Rd		
op																															

Vector

```
SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);    Rmhi = M;
    otherwise UNDEFINED;

integer d = otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQDMULH (vector)

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see [SQRDMULH](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1				Rm			1	0	1	1	0	1								Rd		
U																															

### Scalar

SQDMULH <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			1	0	1	1	0	1								Rd		
U																															

### Vector

SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa **v30.25v29.05**, AdvSIMD **v27.01v26.0**, pseudocode **v85-xml-00bet8\_rc3v35.3**; Build timestamp: **2018-09-13T13:20:04Z**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The `SQDMULL` instruction extracts the first source vector from the lower half of the first source register, while the `SQDMULL2` instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size	L	M				Rm		1	0	1	1	H	0					Rn				Rd	

### Scalar

```
SQDMULL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UNDEFINED;
integer d = otherwise UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M	Rm				1	0	1	1	H	0	Rn				Rd						



Vector

```
SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);    Rmhi = M;
    otherwise UNDEFINED;

integer d = otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>** Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size  $\langle T_s \rangle$  is H.

**<Ts>** Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index>	Is the element index, encoded in “size:L:H:M”:
---------	--

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(datasize)    operand1 = Vpart[n, part];
bits(idxsdszie)   operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2*esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:04Z2018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register, while the SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTN\\_EL2](#), and [CPTN\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1				Rm			1	1	0	1	0	0				Rn				Rd		

### Scalar

SQDMULL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			1	1	0	1	0	0				Rn				Rd		

### Vector

SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

## Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in "size":

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in "size":

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2*esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

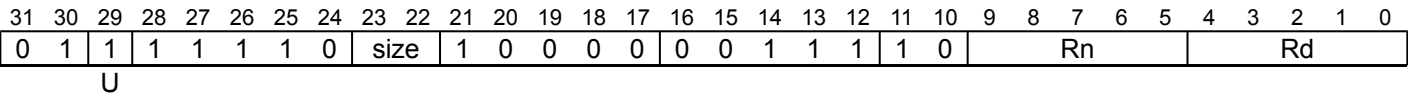
V[d] = result;
```

SQNEG

Signed saturating Negate. This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



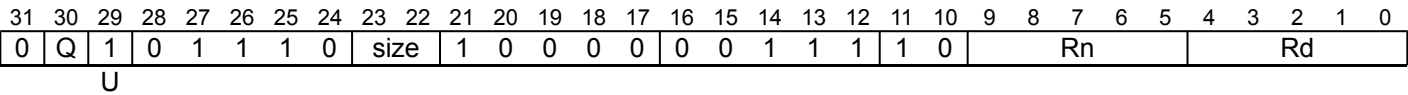
Scalar

```
SQNEG <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector



Vector

```
SQNEG <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

- <V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D
- <d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n>

Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
  element = SInt(Elem[operand, e, esize]);
  if neg then
    element = -element;
  else
    element = Abs(element);
  (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
  if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQRDMLAH (by element)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSR.QC](#), is set if saturation occurs.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	size	L	M			Rm			1	1	0	1	H	0										
																	S														

### Scalar

SQRDMLAH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = () then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UnallocatedEncoding(H:L); Rmhi = M;
  otherwise UNDEFINED;
();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

### Vector (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size	L	M			Rm			1	1	0	1	H	0										
																	S														

Vector

```
SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = () then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding(H:L); Rmhi = M;
    otherwise UNDEFINED;
();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:



size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSR.QC*, is set if saturation occurs.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	0				Rm			1	0	0	0	0	1										
																					S		Rn						Rd		

### Scalar

SQRDMLAH <V><d>, <V><n>, <V><m>

```

if !HaveQRDMLAHExt() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue(Rm);
if size == '11' || size == '00' then UNDEFINED;
();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

### Vector (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	0				Rm			1	0	0	0	0	1										
																					S		Rn						Rd		

Vector

```
SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue(Rm);
if size == '11' || size == '00' then UNDEFINED;
{};
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQRDMLSH (by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSR.QC](#), is set if saturation occurs.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	size	L	M			Rm			1	1	1	1	H	0					Rn				Rd	
																	S														

### Scalar

SQRDMLSH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```

if !HaveQRDMLAHExt() then UNDEFINED;
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = () then UnallocatedEncoding();
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding(H:L); Rmhi = M;
    otherwise UNDEFINED;
();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

```

### Vector (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size	L	M			Rm			1	1	1	1	H	0				Rn					Rd	
																	S														

Vector

```
SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = () then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding(H:L); Rmhi = M;
    otherwise UNDEFINED;
();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SQRDMLSH (vector)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSR.QC*, is set if saturation occurs.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	0				Rm			1	0	0	0	1	1										
																S															

### Scalar

SQRDMLSH <V><d>, <V><n>, <V><m>

```

if !HaveQRDMLAHExt() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue(Rm);
if size == '11' || size == '00' then UNDEFINED;
();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

### Vector (ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	0				Rm			1	0	0	0	1	1										
																S															



Vector

```
SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue(Rm);
if size == '11' || size == '00' then UNDEFINED;
();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQRDMULH (by element)

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see [SQDMULH](#).

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size	L	M				Rm		1	1	0	1	H	0					Rn				Rd	
																op															

### Scalar

SQRDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UNDEFINED;
integer d = otherwise UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M				Rm		1	1	0	1	H	0								Rd		
																op															

Vector

```
SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);    Rmhi = M;
    otherwise UNDEFINED;

integer d = otherwise UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQRDMULH (vector)

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see [SQDMULH](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	1	1	1	1	0	size	1	Rm						1	0	1	1	0	1	Rn						Rd				
U																																

### Scalar

SQRDMULH <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	Q	1	0	1	1	1	0	size	1	Rm						1	0	1	1	0	1	Rn						Rd				
U																																

### Vector

SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:04+00:002018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQRSHL

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see [SQSHL](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	1	1	1	1	0	size	1	Rm						0	1	0	1	1	1	Rn						Rd				
U								R S																								

### Scalar

SQRSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED; if S == '0' && size != '11' then ReservedValue();
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						0	1	0	1	1	1	Rn						Rd			
U										R S																					

### Vector

SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler Symbols

<V> Is a width specifier, encoded in “size”:



size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQRSHRN, SQRSHRN2

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see [SQSHRN](#).

The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_ELI](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	0	0	1	1	1	Rn				Rd					
U									immh				op																		

### Scalar

SQRSHRN <Vb><d>, <Va><n>, #<shift>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	1	1	Rn				Rd					
U									immh				op																		

Vector

```
SQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQRSHRUN, SQRSHRUN2

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see [SQSHRUN](#).

The SQRSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	0	0	0	1	1	Rn				Rd					
immh										op																					

### Scalar

SQRSHRUN <Vb><d>, <Va><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	0	1	1	Rn				Rd					
immh										op																					

Vector

```
SQRSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immh”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immh”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

## Operation

```

checkFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
  element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
  (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
  if sat then FPSR.QC = '1';

Vpart[d, part] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:04Z2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## SQSHL (immediate)

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see [UQRSHL](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	1	1	1	0	1	Rn				Rd					
U				immh								op																			

### Scalar

SQSHL <V><d>, <V><n>, #<shift>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << if immh == '0000' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UnallocatedEncoding(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
  () ;

```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	1	1	1	0	1	Rn				Rd					
U				immh								op																			



Vector

SQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UnallocatedEncoding(immh:immb) - esize;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt (immh:immb)-8)
001x	(UInt (immh:immb)-16)
01xx	(UInt (immh:immb)-32)
1xxx	(UInt (immh:immb)-64)

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:04:45](#)~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQSHL (register)

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see [SQRSHL](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	Rm						0	1	0	0	1	1	Rn						Rd			
U										R										S											

### Scalar

SQSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED; if S == '0' && size != '11' then ReservedValue();
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	Q	0	0	1	1	1	0	size	1	Rm						0	1	0	0	1	1	Rn						Rd				
U										R S																						

### Vector

SQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQSHLU

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see [UQRSHL](#).

If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	1	1	0	0	1	Rn				Rd					
U									immh				op																		

### Scalar

SQSHLU [<V><d>](#), [<V><n>](#), #[<shift>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << if immh == '0000' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UnallocatedEncoding(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	1	0	0	1	Rn				Rd					
U									immh				op																		

Vector

```
SQSHLU <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UnallocatedEncoding(immh:immb) - esize;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler Symbols

- <V>

Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D
- <d>

Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D
- <Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <shift>

For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:04:45](#)~~2018-06-16T09:04:45~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SQSHRN, SQSHRN2

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see [SQRSHRN](#).

The SQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn				Rd					
U									immh				op																		

### Scalar

SQSHRN [<Vb><d>](#), [<Va><n>](#), [#<shift>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn				Rd					
U									immh				op																		



Vector

```
SQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa [v30.25-v29.05](#), AdvSIMD [v27.01-v26.0](#), pseudocode [v85-xml-00bet8\\_rc3-v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

## SQSHRUN, SQSHRUN2

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see [SQSHRUN](#).

The SQSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	0	0	0	0	1	Rn				Rd					
immh												op																			

### Scalar

SQSHRUN [<Vb><d>](#), [<Va><n>](#), [#<shift>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	0	0	1	Rn				Rd					
immh												op																			

Vector

```
SQSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)~~bet8~~\_rc3~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

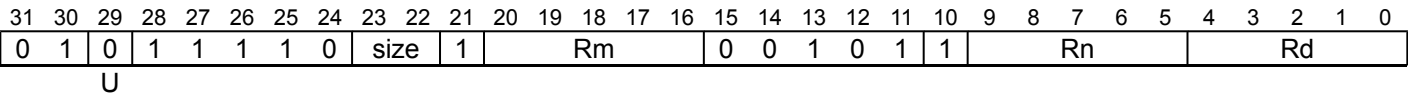
(new)

# SQSUB

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set. Depending on the settings in the *CPACR\_ELI*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

## Scalar

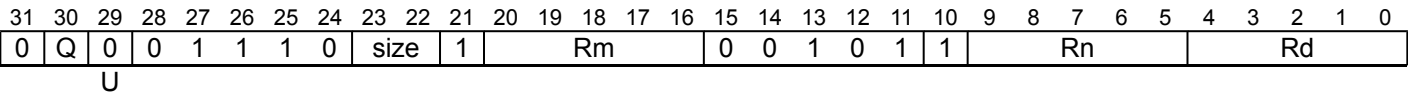


## Scalar

```
SQSUB <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

## Vector



## Vector

```
SQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<I>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## SQXTN, SQXTN2

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The `SQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	0	0	0	0	0	1	0	1	0	0	1	0	Rn					Rd				
U																															

### Scalar

`SQXTN <Vb><d>, <Va><n>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	1	0	1	0	0	1	0										
U																Rn								Rd							

### Vector

`SQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```



Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

## SQXTUN, SQXTUN2

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The `SQXTUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1	0	0	0	0	0	1	0	0	1	0	1	0	Rn					Rd				

### Scalar

`SQXTUN <Vb><d>, <Va><n>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	1	0	0	1	0	1	0	Rn						Rd					

### Vector

`SQXTUN{2} <Vd>.<Tb>, <Vn>.<Ta>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

## Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>	Is an arrangement specifier, encoded in “size:Q”:
------	---

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<Ta>** Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

**<Vb>** Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<b>&lt;Va&gt;</b>	Is the source width specifier, encoded in “size”:
-------------------	---

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htlmdiff from-
```

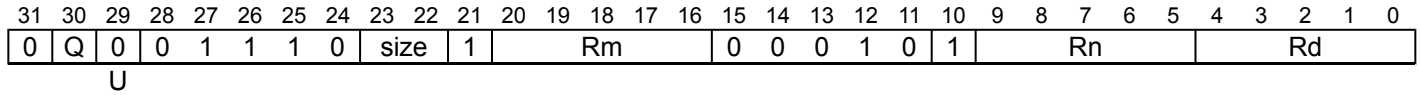
(new)

## SRHADD

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see [SHADD](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

SRHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1 + element2 + 1)<esize:1>;

V[d] = result;
```

(old)

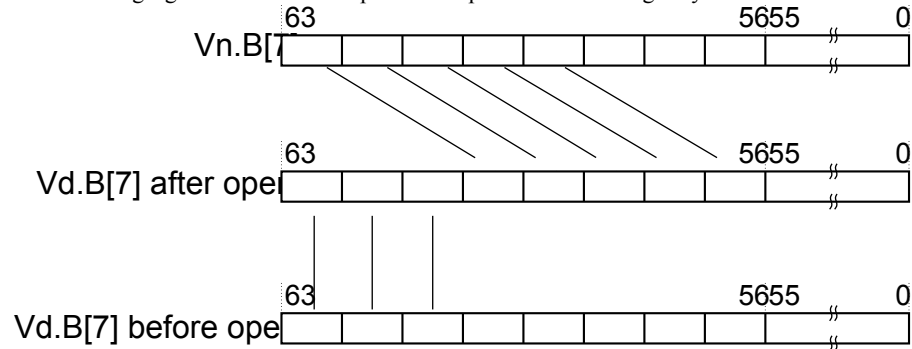
htmldiff from-

(new)

## SRI

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

The following figure shows an example of the operation of shift right by 3 for an 8-bit vector element.



Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000			immb			0	1	0	0	0	1	Rn				Rd						
immh																															

### Scalar

SRI <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) -if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	0	0	0	1	Rn				Rd					
immh																															

Vector

```
SRI <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand  = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
bits(esize) mask = LSR(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSR(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## SRSHL

Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see [SSHL](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					0	1	0	1	0	1	Rn					Rd				
U										R S																					

### Scalar

SRSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED; if S == '0' && size != '11' then ReservedValue();
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm					0	1	0	1	0	1	Rn					Rd					
U										R										S											

### Vector

SRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:18+00:002018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SRRSHR

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see [SSHR](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	0	1	0	0	1	Rn				Rd					
U		immh								o1 o0																					

### Scalar

SRRSHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) -if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	0	1	0	0	1	Rn				Rd					
U				immh								o1 o0																			

Vector

SRSRHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SRSRA

Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see [SSRA](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	0	1	1	0	1	Rn				Rd					
U									immh				o1 o0																		

### Scalar

SRSRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) -if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	0	1	1	0	1	Rn				Rd					
U									immh				o1			o0															

Vector

SRSRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	1	1	1	1	1
																CRm				opc											

## System

SSBB

```

MemBarrierOp op;
MReqDomain domain;
MReqTypes types;

case opc of
  when '00' op = MemBarrierOp_DSB;
  when '01' op = MemBarrierOp_DMB;
  when '10' op = MemBarrierOp_ISB;
  otherwise
    if otherwise HaveSBExtUnallocatedEncoding() && CRm<3:0> == '0000' then
      op = ();
    case CRm<3:2> of
      when '00' domain = MemBarrierOp_SB;
      else
        UNDEFINED;
    case CRm<3:2> of
      when '00' domain = MReqDomain_OuterShareable;
      when '01' domain = MReqDomain_Nonshareable;
      when '10' domain = MReqDomain_InnerShareable;
      when '11' domain = MReqDomain_FullSystem;

case CRm<1:0> of
  when '01' types = MReqTypes_Reads;
  when '10' types = MReqTypes_Writes;
  when '11' types = MReqTypes_All;
  otherwise
    if CRm<3:2> == '01' then
      if CRm<3:2> == '00' then
        op = MemBarrierOp_PSSBB;
      elsif CRm<3:2> == '00' && opc == '00' then
        op = MemBarrierOp_SSBB;
      elsif CRm<3:2> == '01' then
        op = HaveSBExtMemBarrierOp_PSSBB() && CRm<3:2> == '00' && opc == '11' then
          op = MemBarrierOp_SB;
      else
        types = MReqTypes_All;
        domain = MReqDomain_FullSystem;

```

Operation

```
case op of
  when MemBarrierOp_DSBDDataSynchronizationBarrier(domain, types);
  when MemBarrierOp_DMBDataMemoryBarrier(domain, types);
  when MemBarrierOp_ISBInstructionSynchronizationBarrier();
  when MemBarrierOp_SSBBSpeculativeSynchronizationBarrierToVA();
  when MemBarrierOp_PSSBBSpeculativeSynchronizationBarrierToPA();
  when (); MemBarrierOp_SBSpeculationBarrier();
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SSHL

Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see [SRSHL](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	1	1	1	1	0	size	1	Rm						0	1	0	0	0	1	Rn						Rd				
U										R S																						

### Scalar

SSHL [<V><d>](#), [<V><n>](#), [<V><m>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED; if S == '0' && size != '11' then ReservedValue();
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	Q	0	0	1	1	1	0	size	1	Rm						0	1	0	0	0	1	Rn						Rd				
U										R S																						

### Vector

SSHL [<Vd>.<T>](#), [<Vn>.<T>](#), [<Vm>.<T>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

## Assembler Symbols

[<V>](#) Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SSHLL, SSHLL2

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SSHLL instruction extracts vector elements from the lower half of the source register, while the SSHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [SXTL](#), [SXTL2](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	1	0	0	1	Rn				Rd					
U									immh																						

### Vector

```
SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt (immh:immb) - 8)
001x	(UInt (immh:immb) - 16)
01xx	(UInt (immh:immb) - 32)
1xxx	RESERVED

## Alias Conditions

Alias	Is preferred when
<a href="#">SXTL, SXTL2</a>	immb == '000' && <a href="#">BitCount</a> (immh) == 1

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa [v30.25](#)[v29.05](#), AdvSIMD [v27.01](#)[v26.0](#), pseudocode [v85-xml-00](#)[bet8](#) [rc3](#)[v35.3](#); Build timestamp: [2018-09-13](#)[T13](#)[2018-06-16](#)[T09](#)[0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SSHR

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see [SRSHR](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	0	0	0	0	1	Rn				Rd					
U		immh								o1				o0																	

### Scalar

SSHR [<V><d>](#), [<V><n>](#), [#<shift>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) -if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	0	0	0	0	1	Rn				Rd							
U				immh								o1				o0																	

Vector

SSHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SSRA

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see [SRSRA](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	0	0	1	0	1	Rn				Rd					
U		immh								o1 o0																					

### Scalar

SSRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) -if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	0	0	1	0	1	Rn				Rd					
U				immh								o1 o0																			

Vector

SSRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SSUBL, SSUBL2

Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The SSUBL instruction extracts each source vector from the lower half of each source register, while the SSUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			0	0	1	0	0	0										
U								o1																							

### Three registers, not all the same type

```
SSUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SSUBW, SSUBW2

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The SSUBW instruction extracts the second source vector from the lower half of the second source register, while the SSUBW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						0	0	1	1	0	0	Rn						Rd			
U										o1																					

### Three registers, not all the same type

```
SSUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	x	x	1	x	size		Rn				Rt					
L										opcode																					

### One register (opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>]
```

### Two registers (opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

### Three registers (opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

### Four registers (opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm				x	x	1	x	size			Rn				Rt					
L											opcode																				

One register, immediate offset (Rm == 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

- <Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED; otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
            endif
        endfor
    endfor
endfor

if wback then
    if m != 31 then
        offs = X[m];
    endif
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
    endif
endif
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	0	S	size	Rn			Rt							
L										R	opcode																				

### 8-bit (opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>]
```

### 16-bit (opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>]
```

### 32-bit (opcode == 100 && size == 00)

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>]
```

### 64-bit (opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	0	Rm				x	x	0	S	size		Rn				Rt						
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<I>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
    ();
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	size	Rn				Rt						
L										opcode																					

### No offset

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm				1	0	0	0	size			Rn				Rt					
L										opcode																					

### Immediate offset (Rm == 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#16
1	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED; otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
            endif
        endfor
    endfor

if wback then
    if m != 31 then
        offs = X[m];
    endif
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
    endif
endif
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	0	S	size	Rn				Rt						
L										R	opcode																				

### 8-bit (opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

### 16-bit (opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

### 32-bit (opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

### 64-bit (opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	1	Rm				x	x	0	S	size	Rn				Rt							
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
    ();
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ST2G

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	0	1	imm9									0	1	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)	(1)	(1)	(1)	(1)

### Post-index

```
ST2G [<Xn|SP>], #<simm>
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
boolean zero_data = FALSE;
```

### Pre-index

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	0	1	imm9									1	1	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)	(1)	(1)	(1)	(1)

### Pre-index

```
ST2G [<Xn|SP>, #<simm>]!
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
boolean zero_data = FALSE;
```

### Signed offset

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	0	1	imm9									1	0	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)(1)(1)(1)(1)				

### Signed offset

```
ST2G [<Xn|SP>{, #<simm>}]
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
boolean zero_data = FALSE;
```



Assembler Symbols

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<sim>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;
bits(4) tag;

SetNotTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if zero_data then
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8*TAG_GRANULE);
    Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(8*TAG_GRANULE);

tag = AllocationTagFromAddress(address);
MemTag[address] = tag;
MemTag[address+TAG_GRANULE] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04  
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file	htmldiff from-	(new)
-------------	----------------	-------

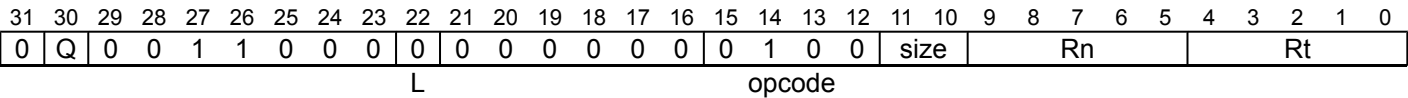
## ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

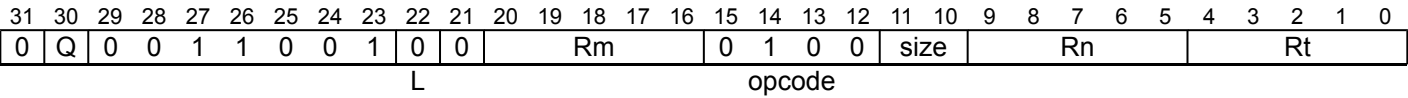


### No offset

ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index



### Immediate offset (Rm == 11111)

ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

### Register offset (Rm != 11111)

ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

- <Vt3>Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm>Is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#24
1	#48
- <Xm>Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED; otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
            endif
        endfor
    endfor

if wback then
    if m != 31 then
        offs = X[m];
    endif
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
    endif
endif
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	1	S	size	Rn				Rt						
L R										opcode																					

### 8-bit (opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>]
```

### 16-bit (opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>]
```

### 32-bit (opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>]
```

### 64-bit (opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	0	Rm				x	x	1	S	size	Rn				Rt							
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>], #3
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B } [<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>], #6
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H } [<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>], #12
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S } [<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>], #24
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D } [<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```

integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
    ();
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	size											
L										opcode																					

### No offset

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm					0	0	0	0	size	Rn					Rt					
L										opcode																					

### Immediate offset (Rm == 11111)

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

### Register offset (Rm != 11111)

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "Q":
 

Q	<imm>
0	#32
1	#64
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED; otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
            endif
        endfor
    endfor

if wback then
    if m != 31 then
        offs = X[m];
    endif
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
    endif
endif
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ST4 (single structure)

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers.

Depending on the settings in the [CPACR\\_ELI](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	1	S	size	Rn				Rt						
L										R										opcode											

### 8-bit (opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

### 16-bit (opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

### 32-bit (opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

### 64-bit (opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	1	Rm					x	x	1	S	size	Rn					Rt					
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = if L == '0' || S == '1' then UnallocatedEncoding();
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = if size<0> == '1' then UnallocatedEncoding();
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = if size<1> == '1' then UnallocatedEncoding();
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UnallocatedEncoding(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
    ();
    index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!wback && n == 31);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## STG

Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	0	1	imm9									0	1	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)(1)(1)(1)(1)				

### Post-index

```
STG [<Xn|SP>], #<simm>
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
boolean zero_data = FALSE;
```

### Pre-index (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	0	1	imm9									1	1	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)	(1)	(1)	(1)	(1)

### Pre-index

```
STG [<Xn|SP>, #<simm>]!
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
boolean zero_data = FALSE;
```

### Signed offset (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	0	1	imm9									1	0	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)	(1)	(1)	(1)	(1)

### Signed offset

```
STG [<Xn|SP>{, #<simm>}]
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
boolean zero_data = FALSE;
```



Assembler Symbols

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simn>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;

SetNotTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if zero_data then
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

MemTag[address] = AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04  
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file	htmldiff from-	(new)
-------------	----------------	-------

## STGP

Store Allocation Tag and Pair of registers stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

### Post-index

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	0	1	0	simm7							Xt2				Xn				Xt						

### Post-index

```
STGP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	1	0	simm7							Xt2				Xn				Xt						

### Pre-index

```
STGP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	0	0	simm7							Xt2				Xn				Xt						

### Signed offset

```
STGP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

## Assembler Symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Xt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Xt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<imm>	For the post-index and pre-index variant: is the signed immediate offset, in the range -64 to 63, encoded in the "simm7" field.  For the signed offset variant: is the optional signed immediate offset, in the range -64 to 63, defaulting to 0 and encoded in the "simm7" field.

## Operation

```
bits(64) address;
bits(64) data1;
bits(64) data2;

SetNotTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data1 = X[t];
data2 = X[t2];

if !postindex then
    address = address + offset;

Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;

MemTag[address] = AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file

htmldiff from-

(new)

## STGV

Store Tag Vector reads from the second source register an IMPLEMENTATION DEFINED number of Allocation Tags and stores them to the naturally aligned array of 16 allocation tags which includes a tag whose address is the address in the first source register. The Allocation Tag at the address in the first source register is always stored, and the first source register is updated to the address of the first Allocation Tag at an address higher than the original address that was not loaded.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

### Integer (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	Xn						Xt			

### Integer

```
STGV <Xt>, [<Xn|SP>]!
```

```
integer t = UInt(Xt);
integer n = UInt(Xn);
```

### Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

### Operation

```
bits(64) data = X[t];
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// address<63:LOG2_TAG_GRANULE+4>:Zeros(LOG2_TAG_GRANULE+4) <= start <= address
// 0 < count <= 16-start<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>
integer count;
(address, count) = ImpDefTagArrayStartAndCount(address);

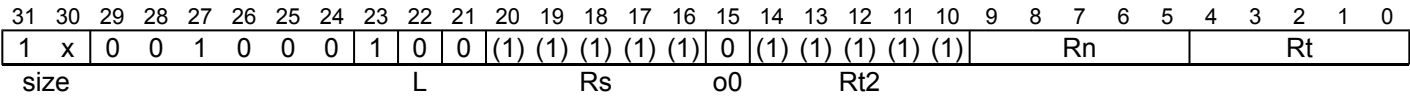
for i = 0 to count-1
    integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);
    bits(4) tag = data<(index*4)+3:index*4>;
    MemTag[address] = tag;
    address = address + TAG_GRANULE;

if n == 31 then
    SP[] = address;
else
    X[n] = address;
```

# STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

## No offset (ARMv8.1)



## 32-bit (size == 10)

```
STLLR <Wt>, [<Xn|SP>{, #0}]
```

## 64-bit (size == 11)

```
STLLR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

```

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00~~bet8\_rc3~~v35.3~~; Build timestamp: ~~2018-09-13T13:20:04~~2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in [Load LOAcquire, Store LORelease](#). For information about memory accesses, see [Load/Store addressing modes](#).

### No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size				L					Rs					o0			Rt2														

### No offset

```
STLLRB <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

(old)

htmldiff from-

(new)



## STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in [Load LOAcquire, Store LORelease](#). For information about memory accesses, see [Load/Store addressing modes](#).

### No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size		L							Rs					o0			Rt2														

### No offset

```
STLLRH <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release
```

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

(old)

htmldiff from-

(new)

## STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size								L			Rs			o0			Rt2														

### 32-bit (size == 10)

```
STLR <Wt>, [<Xn|SP>{, #0}]
```

### 64-bit (size == 11)

```
STLR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType LIMITEDORDERED else AccType ORDERED;
MemOp memop = if L == '1' then MemOp LOAD else MemOp STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

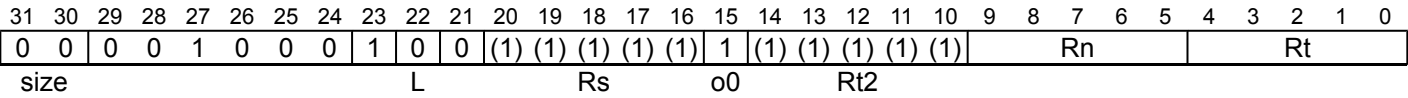
Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).



No offset

```
STLRB <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

(old)

htmldiff from-

(new)

## STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn				Rt					
size		L							Rs				o0			Rt2															

### No offset

```
STLRH <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release
```

```
AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

### Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

(old)

htmldiff from-

(new)

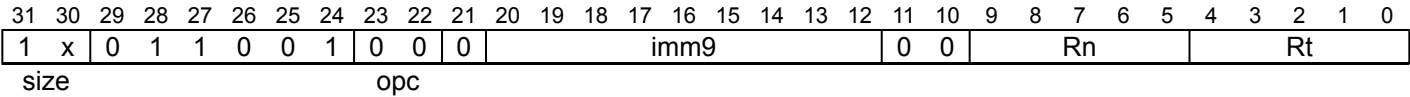


# STLUR

Store-Release Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

The instruction has memory ordering semantics as described in [Load-Acquire, Load-AcquirePC, and Store-Release](#)

For information about memory accesses, see [Load/Store addressing modes](#).



## 32-bit (size == 10)

```
STLUR <Wt>, [<Xn|SP>{, #<simm>}]
```

## 64-bit (size == 11)

```
STLUR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STLURB

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in [Load-Acquire, Load-AcquirePC, and Store-Release](#)

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	0	1	1	0	0	1	0	0	0	imm9									0	0	Rn				Rt									
size								opc																											

### Unscaled offset

```
STLURB <Wt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE;          // writeback is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE;           // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE;          // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding UNDEFINED;
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```



```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STLURH

Store-Release Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in [Load-Acquire, Load-AcquirePC, and Store-Release](#)

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	1	0	0	0	imm9									0	0	Rn				Rt					
size								opc																							

### Unscaled offset

```
STLURH <Wt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	sz	0	0	1	0	0	0	0	0	1	Rs				1	Rt2				Rn				Rt							
L										o0																					

### 32-bit (sz == 0)

```
STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

### 64-bit (sz == 1)

```
STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release
```

```
AccType acctype = if o0 == '1' then AccType_ORDEREDATOMICAccType_ORDERED else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXP](#).

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.

- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.





```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

() ;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	0	0	Rs				1	(1)	(1)	(1)	(1)	(1)	Rn				Rt						
size								L				o0				Rt2															

### 32-bit (size == 10)

```
STLXR <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

### 64-bit (size == 11)

```
STLXR <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC AccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXR](#).

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
        if s == n && n != 31 then
            Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
            assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
            case c of
                when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
                when Constraint_NONE        rn_unknown = FALSE;   // address is original base
                when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
                when Constraint_NOP        EndOfInstruction();

            when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	Rs				1	(1)	(1)	(1)	(1)	(1)	Rn				Rt						
size								L				o0				Rt2															

### No offset

```
STLXRB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType ORDEREDATOMIC else AccType ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXRB](#).

### Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

#### Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.





```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

    when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	Rs				1	(1)	(1)	(1)	(1)	(1)	Rn				Rt						
size								L				o0				Rt2															

### No offset

```
STLXRH <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType ORDEREDATOMIC else AccType ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXRH](#).

### Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

#### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF        UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP          EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF        UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP          EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF        UnallocatedEncoding UNDEFINED;
            when Constraint_NOP          EndOfInstruction();

() ;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## STNP (SIMD&FP)

Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	0	0	0	imm7							Rt2					Rn				Rt					
L																															

### 32-bit (opc == 00)

```
STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

## Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.  For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VECSTREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == t1 && t == t2 then MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;        // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
    end
end

when Constraint_NOP      EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0      , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0      , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        end
        V[t]  = data1;
        V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see [Load/Store addressing modes](#). For information about Non-temporal pair instructions, see [Load/Store Non-temporal pair](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
x	0	1	0	1	0	0	0	0	0	imm7							Rt2				Rn				Rt										
opc										L																									

### 32-bit (opc == 00)

STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]

### 64-bit (opc == 10)

STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UNDEFINED;
integer scale = 2 +if opc<0> == '1' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == && t == t2 then MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;        // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding    UNDEFINED;
    ();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0      , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0      , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        X[t]  = data1;
        X[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
opc		1		0		1		1		0		0		1		0		imm7							Rt2					Rn					Rt			
L																																						

L

#### 32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>], #<imm>
```

#### 64-bit (opc == 01)

```
STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

#### 128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	1	1	0	imm7							Rt2					Rn					Rt				

L

#### 32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

#### 64-bit (opc == 01)

```
STP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

#### 128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	1	0	0	imm7							Rt2					Rn					Rt				

L

### 32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
STP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;
```

## Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	<p>For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as &lt;imm&gt;/16.</p> <p>For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/16.</p>

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer t2 = UInt(Rt2);  
AccType acctype = AccType_VEC;  
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;  
if opc == '11' then UNDEFINED;  
integer scale = 2 + if opc == '11' then UnallocatedEncoding();  
integer scale = 2 + UInt(opc);  
integer datasize = 8 << scale;  
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

if memop == t1 && t == t2 then MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;        // result is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding        UNDEFINED;
    end
end

when Constraint_NOP                EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0, dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0, dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        end
        V[t] = data1;
        V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## STP

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	0	1	0	1	0	0	0	1	0	imm7							Rt2				Rn				Rt						
opc										L																					

#### 32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

#### 64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x 0		1 0 1		0		0 1 1		0		imm7							Rt2				Rn				Rt						
opc										L																					

#### 32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

#### 64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
x		0		1		0		1		0		0		1		0		0		imm7							Rt2				Rn				Rt			
opc										L																												

#### 32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

#### 64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	<p>For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/8.</p>

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```





```

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

boolean wb_unknown = FALSE;

if memop == && wback && (t == n || t2 == n) && n != 31 then MemOp_LOAD && wback && (t == n || t2 == n) &
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE; // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UnallocatedEncoding UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0, dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address + 0, dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;

```

```

    if signed then
        X[t]  = SignExtend(data1, 64);
        X[t2] = SignExtend(data2, 64);
    else
        X[t]  = data1;
        X[t2] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	0	0	imm9									0	1	Rn				Rt					
opc																															

#### 8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>], #<sim>
```

#### 16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>], #<sim>
```

#### 32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>], #<sim>
```

#### 64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>], #<sim>
```

#### 128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	0	0	imm9									1	1	Rn				Rt					
opc																															

8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>, #<sim>]!
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, #<sim>]!
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, #<sim>]!
```

64-bit (size == 11 && opc == 00)

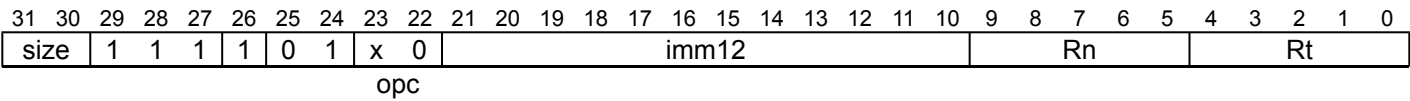
```
STR <Dt>, [<Xn|SP>, #<sim>]!
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>.size);
if scale > 4 then UNDEFINED;
bits(64) offset = if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>.size);
if scale > 4 then UNDEFINED;
bits(64) offset = if scale > 4 then UnallocatedEncoding();
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	<p>For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.</p> <p>For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/2.</p> <p>For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/4.</p> <p>For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/8.</p> <p>For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/16.</p>

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	0	0	imm9									0	1	Rn				Rt					
size										opc																					

#### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>], #<sim>
```

#### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn				Rt					
size										opc																					

#### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, #<sim>]!
```

#### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
1		x		1		1		1		0		0		1		0		0		imm12												Rn					Rt				
size										opc																															

### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>{, #<pimm>}]
```

### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;  
integer scale = UInt(size);  
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	<p>For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/4.</p> <p>For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as &lt;pimm&gt;/8.</p>

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
AccType acctype = AccType.NORMAL;  
MemOp memop;  
boolean signed;  
integer regsize;  
  
if opc<1> == '0' then  
    // store or zero-extending load  
    memop = if opc<0> == '1' then MemOp.LOAD else MemOp.STORE;  
    regsize = if size == '11' then 64 else 32;  
    signed = FALSE;  
else  
    if size == '11' then  
        UNDEFINED;  
    else  
        // sign-extending load  
        memop = if size == '11' then UnallocatedEncoding();  
    else  
        // sign-extending load  
        memop = MemOp.LOAD;  
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();  
        if size == '10' && opc<0> == '1' then UNDEFINED;  
    ();  
    regsize = if opc<0> == '1' then 32 else 64;  
    signed = TRUE;  
  
integer datasize = 8 << scale;
```





```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STR (register, SIMD&FP)

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
size		1		1		1		1		0		0		x		0		1		Rm						option		S		1		0		Rn						Rt		
opc																																										

### 8-fsreg,STR-8-fsreg (size == 00 && opc == 00 && option != 011)

```
STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### 8-fsreg,STR-8-fsreg (size == 00 && opc == 00 && option == 011)

```
STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

### 16-fsreg,STR-16-fsreg (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 32-fsreg,STR-32-fsreg (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-fsreg,STR-64-fsreg (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 128-fsreg,STR-128-fsreg (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED; // sub-word indexif scale > 4 then
UnallocatedEncoding();
if option<1> == '0' then UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#4

Shared Decode

```
integer n = UInt (Rn);
integer t = UInt (Rt);
integer m = UInt (Rm);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if(m, extend_type, shift); HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

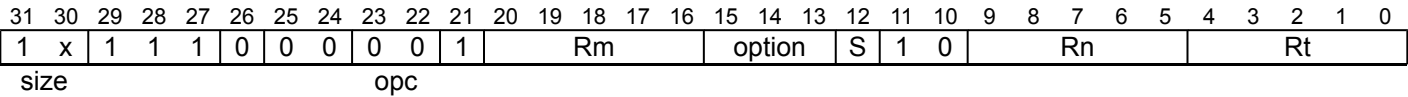
(old)

htmldiff from-

(new)

## STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see [Load/Store addressing modes](#). The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word indexif option<1> == '0' then
UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm>

When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm>

When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend>

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

- <amount>

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

- For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
        else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding;
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```





```

bits(64) offset = ExtendReg(m, extend_type, shift);
if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding  UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	0	0	imm9									0	1	Rn				Rt					
size									opc																						

### Post-index

```
STRB <Wt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
0	0	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn				Rt										
size											opc																									

### Pre-index

```
STRB <Wt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
0		0		1		1		1		0		0		1		0		0		imm12											Rn				Rt			
size										opc																												

### Unsigned offset

```
STRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STRB \(immediate\)](#).

## Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;

```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

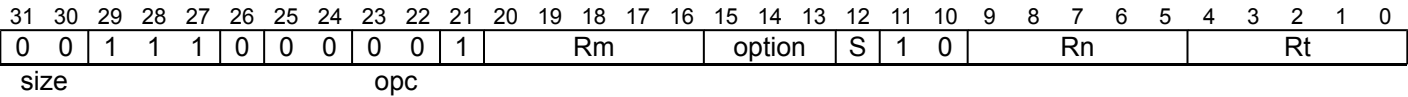
(old)	htmldiff from-	(new)
-------	----------------	-------



## STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see [Load/Store addressing modes](#).

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



### Extended register (option != 011)

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

### Shifted register (option == 011)

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm>

When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm>

When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend>

Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

- <amount>

Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

bits(64) offset = ExtendReg(m, extend_type, shift);
if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    e = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding    UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	0	0	imm9										0	1	Rn					Rt							
size										opc																									

### Post-index

```
STRH <Wt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	0	0	0	imm9										1	1	Rn					Rt							
size										opc																									

### Pre-index

```
STRH <Wt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
0		1		1		1		0		0		1		0		0		imm12														Rn					Rt			
size										opc																														

### Unsigned offset

```
STRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STRH \(immediate\)](#).

## Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = if size == '11' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;

```





```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

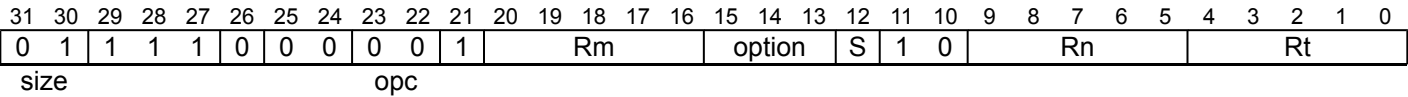
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see [Load/Store addressing modes](#).

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



32-bit

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word indexif option<1> == '0' then
UnallocatedEncoding(); // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

- <Wt>Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm>When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm>When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend>Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

- <amount>Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

bits(64) offset = ExtendReg(m, extend_type, shift);
if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;      // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding  UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

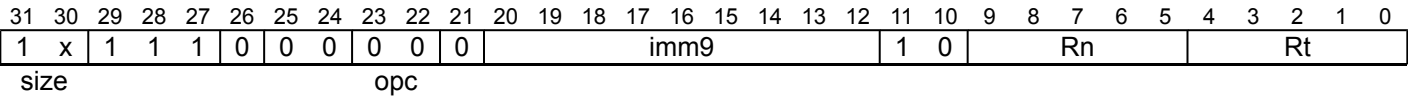
# STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR\_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



## 32-bit (size == 10)

```
STTR <Wt>, [<Xn|SP>{, #<sim>}]
```

## 64-bit (size == 11)

```
STTR <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.



## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == (Rt); EL1AccType && !(acctype == EL2EnabledAccType_UNPRIV() &&
if ( HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = ( && HaveEL(EL2) && ) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
acctype = HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
acctype = AccType_UNPRIV;
else
acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = if size == '11' then 64 else 32;
signed = FALSE;
else
if size == '11' then
UNDEFINED;
else
// sign-extending load
memop = if size == '11' then UnallocatedEncoding();
else
// sign-extending load
memop = MemOp_LOAD;
if size == '10' && opc<0> == '1' then UnallocatedEncoding;
if size == '10' && opc<0> == '1' then UNDEFINED;
();

regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

integer datasize = 8 << scale;

```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE;          // writeback is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when UnallocatedEncoding();
        when Constraint NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint NONE rt_unknown = FALSE;           // value stored is original value
        when Constraint UNKNOWN rt_unknown = TRUE;          // value stored is UNKNOWN
        when Constraint UNDEF UnallocatedEncoding UNDEFINED;
    ();
        when Constraint NOP EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

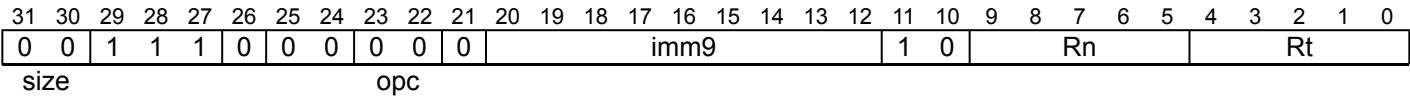
STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR\_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



Unscaled offset

```
STTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == (Rt); EL1AccType && !(acctype == EL2EnabledAccType_UNPRIV() &&
if ( HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = ( && HaveEL(EL2) && ) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
acctype = HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
acctype = AccType_UNPRIV;
else
acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = if size == '11' then 64 else 32;
signed = FALSE;
else
if size == '11' then
UNDEFINED;
else
// sign-extending load
memop = if size == '11' then UnallocatedEncoding();
else
// sign-extending load
memop = MemOp_LOAD;
if size == '10' && opc<0> == '1' then UnallocatedEncoding;
if size == '10' && opc<0> == '1' then UNDEFINED;
();

regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```



```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

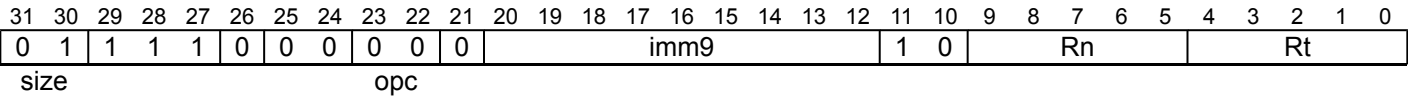
# STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR\_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



## Unscaled offset

```
STTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == (Rt); EL1AccType && !(acctype == EL2EnabledAccType_UNPRIV() &&
if ( HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = ( && HaveEL(EL2) && ) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
acctype = HaveVirtHostExt() && PSTATE.EL == EL2 && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
acctype = AccType_UNPRIV;
else
acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = if size == '11' then 64 else 32;
signed = FALSE;
else
if size == '11' then
UNDEFINED;
else
// sign-extending load
memop = if size == '11' then UnallocatedEncoding();
else
// sign-extending load
memop = MemOp_LOAD;
if size == '10' && opc<0> == '1' then UnallocatedEncoding;
if size == '10' && opc<0> == '1' then UNDEFINED;
();

regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

integer datasize = 8 << scale;

```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:20:09.0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	0	0	imm9									0	0	Rn				Rt					
opc																															

### 8-bit (size == 00 && opc == 00)

```
STUR <Bt>, [<Xn|SP>{, #<sim>}]
```

### 16-bit (size == 01 && opc == 00)

```
STUR <Ht>, [<Xn|SP>{, #<sim>}]
```

### 32-bit (size == 10 && opc == 00)

```
STUR <St>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (size == 11 && opc == 00)

```
STUR <Dt>, [<Xn|SP>{, #<sim>}]
```

### 128-bit (size == 00 && opc == 10)

```
STUR <Qt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>.size);
if scale > 4 then UNDEFINED;
bits(64) offset = if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
if HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

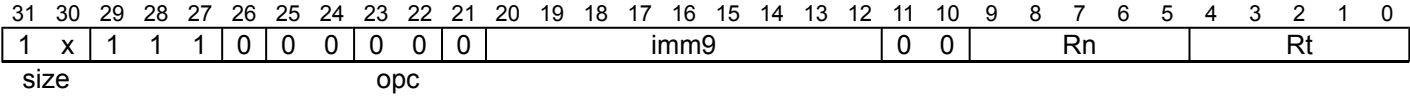
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



# STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see [Load/Store addressing modes](#).



## 32-bit (size == 10)

```
STUR <Wt>, [<Xn|SP>{, #<sim>}]
```

## 64-bit (size == 11)

```
STUR <Xt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	0	0	imm9									0	0	Rn			Rt						
size											opc																				

### Unscaled offset

```
STURB <Wt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```



```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:20:09.0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	0	0	imm9										0	0	Rn					Rt			
size										opc																					

### Unscaled offset

```
STURH <Wt>, [<Xn|SP>{, #<sim>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = if opc<0> == '1' then UnallocatedEncoding();
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UnallocatedEncoding();
        if size == '10' && opc<0> == '1' then UNDEFINED;
    ();
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
```



```

if bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31 && !wback);

bits(64) address;
bits(datasize) data;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == && wback && n == t && n != 31 then
    c = MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;           // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;       // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;       // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
            Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;

```

```
else
    X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

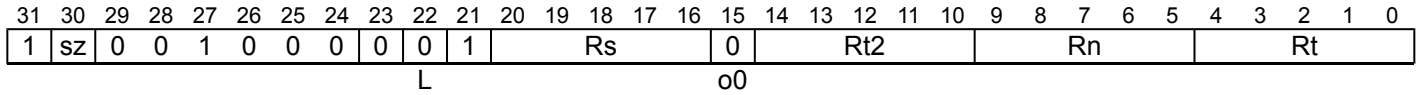
Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STXP

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses see [Load/Store addressing modes](#).



### 32-bit (sz == 0)

```
STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

### 64-bit (sz == 1)

```
STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STXP](#).

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.





```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

    when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0												
1		x		0		0		1		0		0		0		0		0		0		Rs		0		(1)		(1)		(1)		(1)		(1)		Rn				Rt			
size								L								o0								Rt2																			

### 32-bit (size == 10)

```
STXR <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

### 64-bit (size == 11)

```
STXR <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType ORDEREDATOMIC AccType ORDERED else AccType ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp LOAD else MemOp STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STXR](#).

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

    when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
        else // elsize == 64
            // 64-bit load exclusive pair (not atomic),
            // but must be 128-bit aligned
            if address != Align(address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
            X[t] = Mem[address + 0, 8, acctype];
            X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic.

For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	Rs				0	(1)	(1)	(1)	(1)	(1)	Rn				Rt						
size								L				o0				Rt2															

### No offset

```
STXRB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release
```

```
AccType acctype = if o0 == '1' then AccType_ORDEREDATOMICAccType_ORDERED else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STXRB](#).

### Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

#### Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.





```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF        UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP          EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF        UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP          EndOfInstruction();
        if s == n && n != 31 then
            Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
            assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
            case c of
                when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
                when Constraint_NONE        rn_unknown = FALSE;   // address is original base
                when Constraint_UNDEF        UnallocatedEncoding() UNDEFINED;
                when Constraint_NOP          EndOfInstruction();

            );

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic.

For information about memory accesses see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	Rs				0	(1)	(1)	(1)	(1)	(1)	Rn				Rt						
size		L								o0		Rt2																			

### No offset

```
STXRH <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

### Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0	If the operation updates memory.
1	If the operation fails to update memory.
<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

#### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



```

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == HaveMTEExt() then
    boolean is_load_store = memop IN {MemOp_STORE, MemOp_LOAD};
    SetNotTagCheckedInstruction(is_load_store && n == 31);

if memop == && pair && t == t2 then MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when UnallocatedEncoding() ;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE        rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when UnallocatedEncoding() ;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE        rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UnallocatedEncoding() UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

() ;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) e11 = X[t];
            bits(datasize DIV 2) e12 = X[t2];
            data = if BigEndian() then e11 : e12 else e12 : e11;
        else
            data = X[t];

    bit status = '1';
    // Check whether the Exclusive monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, acctype] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD

```

```

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case
        X[t] = bits(datasize) UNKNOWN;
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, dbytes, acctype];
        if BigEndian() then
            X[t] = data<datasize-1:elsize>;
            X[t2] = data<elsize-1:0>;
        else
            X[t] = data<elsize-1:0>;
            X[t2] = data<datasize-1:elsize>;
    else // elsize == 64
        // 64-bit load exclusive pair (not atomic),
        // but must be 128-bit aligned
        if address != Align(address, dbytes) then
            iswrite = FALSE;
            secondstage = FALSE;
            AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
        X[t] = Mem[address + 0, 8, acctype];
        X[t2] = Mem[address + 8, 8, acctype];
    else
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00~~~~bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## STZ2G

Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	1	1	imm9									0	1	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)	(1)	(1)	(1)	

### Post-index

```
STZ2G [<Xn|SP>], #<sim>
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
boolean zero_data = TRUE;
```

### Pre-index

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	1	1	imm9									1	1	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)	(1)	(1)	(1)	(1)

### Pre-index

```
STZ2G [<Xn|SP>, #<sim>]!
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
boolean zero_data = TRUE;
```

### Signed offset

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	1	1	imm9									1	0	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)	(1)	(1)	(1)	(1)

### Signed offset

```
STZ2G [<Xn|SP>{, #<sim>}]
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
boolean zero_data = TRUE;
```

Assembler Symbols

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<sim>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;
bits(4) tag;

SetNotTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if zero_data then
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8*TAG_GRANULE);
    Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(8*TAG_GRANULE);

tag = AllocationTagFromAddress(address);
MemTag[address] = tag;
MemTag[address+TAG_GRANULE] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04  
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file	htmldiff from-	(new)
-------------	----------------	-------



## STZG

Store Allocation Tag, Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the source register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#), [Pre-index](#) and [Signed offset](#)

### Post-index

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	1	1	imm9									0	1	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)	(1)	(1)	(1)	(1)

### Post-index

```
STZG [<Xn|SP>], #<simm>
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
boolean zero_data = TRUE;
```

### Pre-index

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	1	1	imm9									1	1	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)	(1)	(1)	(1)	

### Pre-index

```
STZG [<Xn|SP>, #<simm>]!
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
boolean zero_data = TRUE;
```

### Signed offset

(ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	1	1	imm9									1	0	Xn			(1)	(1)	(1)	(1)	(1)		
																											(1)	(1)	(1)	(1)	(1)

### Signed offset

```
STZG [<Xn|SP>{, #<simm>}]
```

```
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
boolean zero_data = TRUE;
```

Assembler Symbols

<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simn>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;

SetNotTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if zero_data then
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

MemTag[address] = AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file	htmldiff from-	(new)
-------------	----------------	-------

## SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	0	1	0	1	1	0	0	1	Rm					option			imm3			Rn			Rd						
op		S																													

### 32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

### 64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED; if shift > 4 then ReservedValue();
```

## Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

```

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25-v29.05, AdvSIMD v27.01-v26.0, pseudocode v85-xml-00bet8\_rc3-v35.3; Build timestamp: 2018-09-13T13:20:18-06:16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

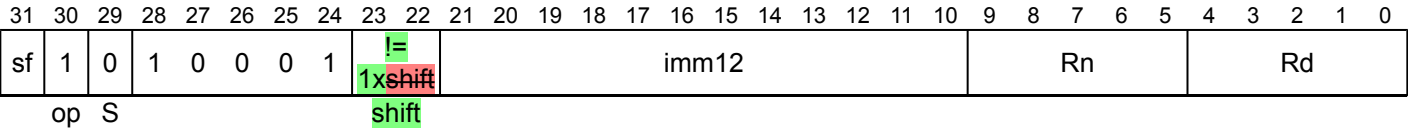
(old)

htmldiff from-

(new)

SUB (immediate)

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.



32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
  when '00' imm = ZeroExtend(imm12, datasize);
  when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
  when '10' SEE "ADDG, SUBG";
  when '11' when '1x' ReservedValue();
```

Assembler Symbols

- <Wd|WSP>

Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Xd|SP>

Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- <shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

shift<0>	<shift>
000	LSL #0
101	LSL #12
1x	RESERVED

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

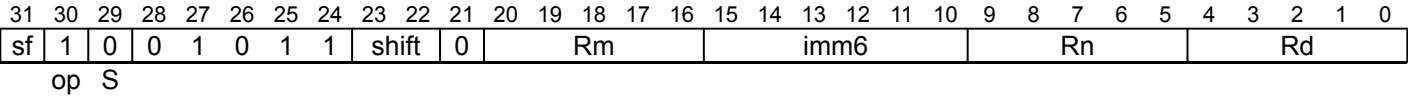
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias [NEG \(shifted register\)](#).



### 32-bit (sf == 0)

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED; if shift == '11' then
ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

## Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">NEG (shifted register)</a>	Rn == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



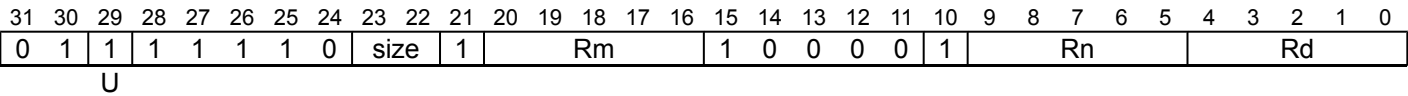
## SUB (vector)

Subtract (vector). This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

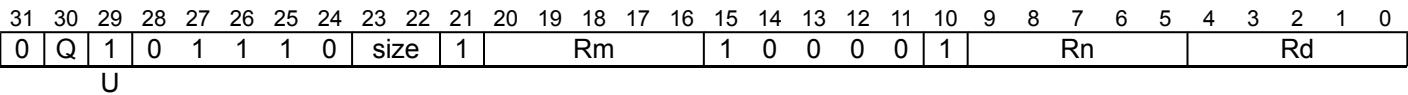


### Scalar

SUB <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

### Vector



### Vector

SUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

### Assembler Symbols

- <V>

Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D
- <d>

Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d] = result;

```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

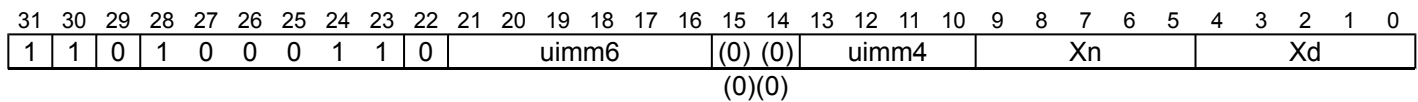
htmldiff from-

(new)

## SUBG

Subtract with Tag subtracts an immediate value scaled by the Tag granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR\_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

### Integer (ARMv8.5)



### Integer

```
SUBG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>
```

```
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(4) tag_offset = uimm4;
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
boolean ADD = FALSE;
```

### Assembler Symbols

<Xd SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
<Xn SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.
<uimm6>	Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
<uimm4>	Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

### Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(4) start_tag = AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AllocationTagAccessIsEnabled() then
    rtag = ChooseNonExcludedTag(start_tag, tag_offset, exclude);
else
    rtag = '0000';

if ADD then
    (result, -) = AddWithCarry(operand1, offset, '0');
else
    (result, -) = AddWithCarry(operand1, NOT(offset), '1');

result = AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

no old file

htmldiff from-

(new)

## SUBHN, SUBHN2

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see [RSUBHN](#).

The SUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			0	1	1	0	0	0										
U										o1																					

### Three registers, not all the same type

```
SUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SUBP

Subtract Pointer subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register.

### Integer (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	1	1	0	Xm					0	0	0	0	0	0	Xn					Xd				

### Integer

```
SUBP <Xd>, <Xn|SP>, <Xm|SP>
```

```
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
boolean setflags = FALSE;
```

### Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
- <Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- <Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

### Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) operand2 = if m == 31 then SP[] else X[m];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;
bits(4) nzcvc;

operand2 = NOT(operand2);
(result, nzcvc) = AddWithCarry(operand1, operand2, '1');

if setflags then
    PSTATE.<N,Z,C,V> = nzcvc;
X[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SUBPS

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias [CMPP](#).

### Integer (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	1	1	0	Xm				0	0	0	0	0	0	Xn				Xd						

### Integer

SUBPS <Xd>, <Xn|SP>, <Xm|SP>

```
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
boolean setflags = TRUE;
```

### Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
<Xn SP>	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
<Xm SP>	Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

### Alias Conditions

Alias	Is preferred when
<a href="#">CMPP</a>	S == '1' && Xd == '11111'

### Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) operand2 = if m == 31 then SP[] else X[m];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;
bits(4) nzcvc;

operand2 = NOT(operand2);
(result, nzcvc) = AddWithCarry(operand1, operand2, '1');

if setflags then
    PSTATE.<N,Z,C,V> = nzcvc;
X[d] = result;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## SUBS (extended register)

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias [CMP \(extended register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	0	1	0	1	1	0	0	1	Rm				option			imm3			Rn				Rd						
op S																															

### 32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

### 64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED; if shift > 4 then ReservedValue();
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn WSP>	Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn SP>	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
<R>	Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m>	Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
<extend>	For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

## Alias Conditions

## Operation

## Operational information

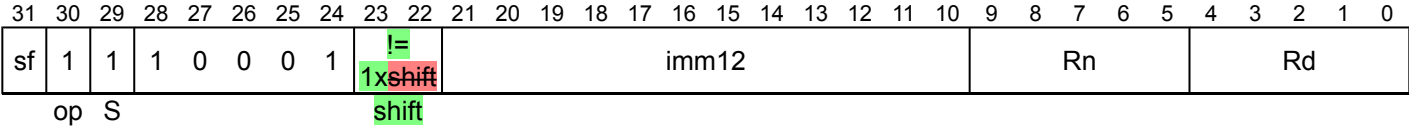
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(new)

SUBS (immediate)

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [CMP \(immediate\)](#).



32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');
bits(datasize) imm;

case shift of
  when '00' imm = ZeroExtend(imm12, datasize);
  when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
  when '10' SEE "ADDG, SUBG";
  when '11' when '1x' ReservedValue();
```

Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- <shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift<0>”:

shift<0>	<shift>
000	LSL #0
101	LSL #12
1x	RESERVED

Alias Conditions

Alias	Is preferred when
<a href="#">CMP (immediate)</a>	Rd == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcv) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

if d == 31 && !setflags then
    SP[] = result;
else
    X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

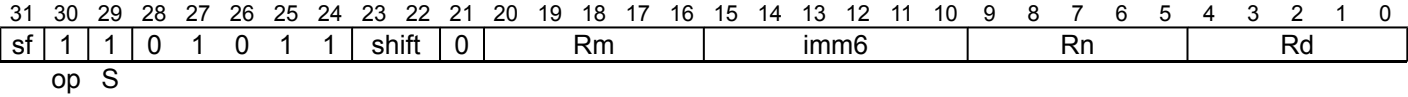
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SUBS (shifted register)

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases [CMP \(shifted register\)](#), and [NEGS](#).



### 32-bit (sf == 0)

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
boolean setflags = (S == '1');

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED; if shift == '11' then
ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

## Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

Alias	Is preferred when
<a href="#">CMP (shifted register)</a>	Rd == '11111'
<a href="#">NEGS</a>	Rn == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcvc;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, carry_in);

if setflags then
    PSTATE.<N,Z,C,V> = nzcvc;

X[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:20:18-06:16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)htmldiff from-(new)

## SUQADD

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	0	0	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
U																															

### Scalar

SUQADD [<V><d>](#), [<V><n>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	1	1	1	0	Rn				Rd					
U																															

### Vector

SUQADD [<Vd>.<T>](#), [<Vn>.<T>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

## Assembler Symbols

[<V>](#) Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



### SWP, SWPA, SWPAL, SWPL

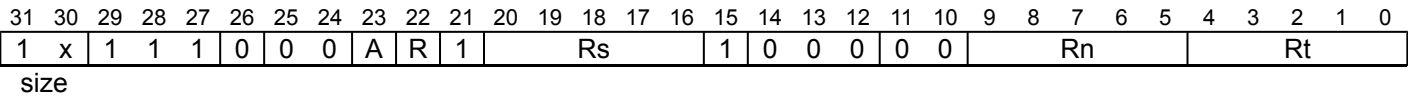
Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, SWPA and SWPAL load from memory with acquire semantics.
- SWPL and SWPAL store to memory with release semantics.
- SWP has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

#### Integer (ARMv8.1)



### 32-bit SWP (size == 10 && A == 0 && R == 0)

```
SWP <Ws>, <Wt>, [<Xn|SP>]
```

### 32-bit SWPA (size == 10 && A == 1 && R == 0)

```
SWPA <Ws>, <Wt>, [<Xn|SP>]
```

### 32-bit SWPAL (size == 10 && A == 1 && R == 1)

```
SWPAL <Ws>, <Wt>, [<Xn|SP>]
```

### 32-bit SWPL (size == 10 && A == 0 && R == 1)

```
SWPL <Ws>, <Wt>, [<Xn|SP>]
```

### 64-bit SWP (size == 11 && A == 0 && R == 0)

```
SWP <Xs>, <Xt>, [<Xn|SP>]
```

### 64-bit SWPA (size == 11 && A == 1 && R == 0)

```
SWPA <Xs>, <Xt>, [<Xn|SP>]
```

### 64-bit SWPAL (size == 11 && A == 1 && R == 1)

```
SWPAL <Xs>, <Xt>, [<Xn|SP>]
```

### 64-bit SWPL (size == 11 && A == 0 && R == 1)

```
SWPL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t =() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType atomicrw = if R == '1' then AccType_ATOMICRW else AccType_ATOMICRW;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
Mem[address, datasize DIV 8, stacctype] = X[s];

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SWPB, SWPAB, SWPALB, SWPLB

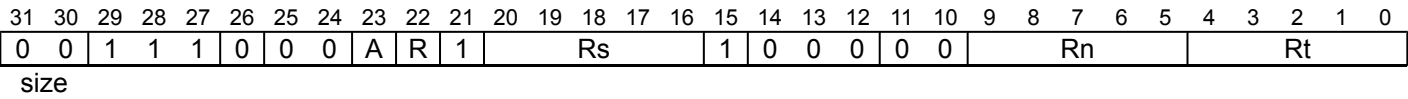
Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer (ARMv8.1)



#### SWPAB (A == 1 && R == 0)

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

#### SWPALB (A == 1 && R == 1)

```
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

#### SWPB (A == 0 && R == 0)

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

#### SWPLB (A == 0 && R == 1)

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = () then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

### Assembler Symbols

- <Ws>

Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- <Wt>

Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
Mem[address, datasize DIV 8, stacctype] = X[s];

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has no memory ordering requirements.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					1	0	0	0	0	0	Rn					Rt				
size																															

#### SWPAH (A == 1 && R == 0)

```
SWPAH <Ws>, <Wt>, [<Xn|SP>]
```

#### SWPALH (A == 1 && R == 1)

```
SWPALH <Ws>, <Wt>, [<Xn|SP>]
```

#### SWPH (A == 0 && R == 0)

```
SWPH <Ws>, <Wt>, [<Xn|SP>]
```

#### SWPLH (A == 0 && R == 1)

```
SWPLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;
integer t = () then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDATOMICRW else AccType_ORDEREDRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

### Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;

if n == 31 then HaveMTEExt() then
    SetNotTagCheckedInstruction(n == 31);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];
Mem[address, datasize DIV 8, stacctype] = X[s];

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

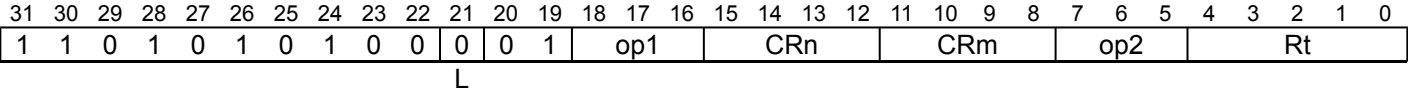
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

SYS

System instruction. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.

This instruction is used by the aliases [AT](#), [CFP](#), [CPP](#), [DC](#), [DVP](#), [IC](#), and [TLBI](#).



System

SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}

```
AArch64.CheckSystemAccess ('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 1;
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
boolean has_result = (L == '1');
```

Assembler Symbols

- <op1>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn>

Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm>

Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- <Xt>

Is the 64-bit name of the optional general-purpose source register, defaulting to '1111', encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
<a href="#">AT</a>	CRn == '0111' && CRm == '100x' && <a href="#">SysOp</a> (op1, '0111', CRm, op2) == <a href="#">Sys_AT</a>
<a href="#">CFP</a> <a href="#">DC</a>	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '100' <a href="#">SysOp</a> (op1, '0111', CRm, op2) == <a href="#">Sys_CFP</a> <a href="#">Sys_DC</a>
<a href="#">CPP</a> <a href="#">IC</a>	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '111' <a href="#">SysOp</a> (op1, '0111', CRm, op2) == <a href="#">Sys_CPP</a> <a href="#">Sys_IC</a>
<a href="#">DC</a> <a href="#">TLBI</a>	CRn == '01111000' && <a href="#">SysOp</a> (op1, '01111000', CRm, op2) == <a href="#">Sys_DC</a> <a href="#">Sys_TLBI</a>
<a href="#">DVP</a>	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '101'
<a href="#">IC</a>	CRn == '0111' && <a href="#">SysOp</a> (op1, '0111', CRm, op2) == <a href="#">Sys_IC</a>
<a href="#">TLBI</a>	CRn == '1000' && <a href="#">SysOp</a> (op1, '1000', CRm, op2) == <a href="#">Sys_TLBI</a>

Operation

```
if has_result then
    X[t] = AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    AArch64.SysInstr(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```



(old)

htmldiff from-

(new)

## TBL

Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	Rm				0	len	0	0	0	Rn				Rd							
op																															

### Two register table (len == 01)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

### Three register table (len == 10)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

### Four register table (len == 11)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

### Single register table (len == 00)

```
TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	8B
1	16B

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;
integer i;

// Create table from registers
for i = 0 to regs - 1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements - 1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

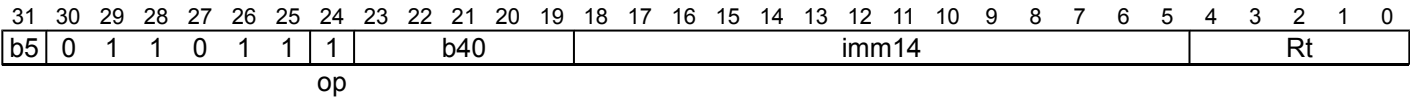
Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



14-bit signed PC-relative branch offset

```
TBNZ <R><t>, #<imm>, <label>
```

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bit bit_val = op;
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

- <R>

Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.
- <t>

Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.
- <imm>

Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".
- <label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == bit_val then
  BranchTo(PC[] + offset, BranchType_DIRBranchType_JMP);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## TBX

Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	Rm				0	len	1	0	0	Rn				Rd							
op																															

### Two register table (len == 01)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

### Three register table (len == 10)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

### Four register table (len == 11)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

### Single register table (len == 00)

```
TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	8B
1	16B

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;
integer i;

// Create table from registers
for i = 0 to regs - 1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements - 1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

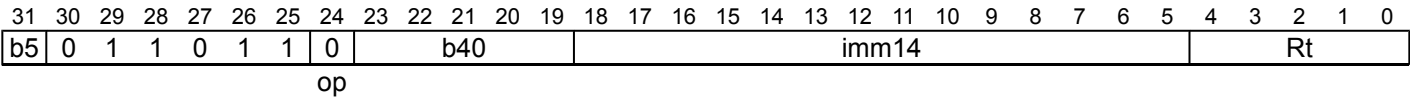
Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



14-bit signed PC-relative branch offset

```
TBZ <R><t>, #<imm>, <label>

integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bit bit_val = op;
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

- <R>

Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.
- <t>

Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.
- <imm>

Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".
- <label>

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == bit_val then
    BranchTo(PC[] + offset, BranchType_DIRBranchType_JMP);
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

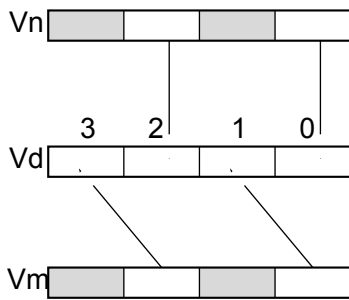
## TRN1

Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

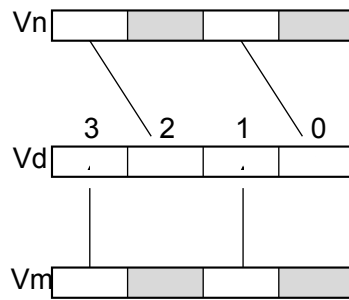
By using this instruction with TRN2, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.

TRN1.16



TRN2.16



Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	0	size	0	Rm						0	0	1	0	1	0	Rn						Rd					
op																																	

### Advanced SIMD

TRN1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;

```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.



Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer p;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

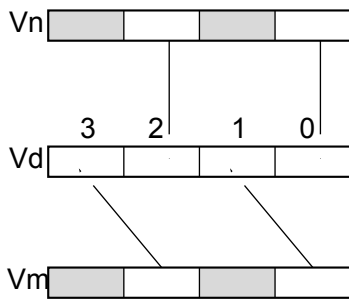
## TRN2

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

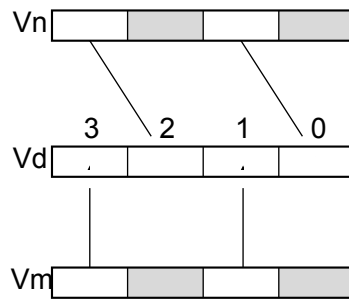
By using this instruction with TRN1, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.

TRN1.16



TRN2.16



Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	0	size	0	Rm						0	1	1	0	1	0	Rn						Rd					
op																																	

### Advanced SIMD

TRN2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;

```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer p;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions.

If the Self-Hosted Trace Extension is not implemented, this instruction executes as a NOP.

### System (ARMv8.4)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	1	0	1	1	1	1	1							
																				CRm						op2												

### System

TSB CSYNC

```

SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp NOP;
  when '0000 001' op = SystemHintOp YIELD;
  when '0000 010' op = SystemHintOp WFE;
  when '0000 011' op = SystemHintOp WFI;
  when '0000 100' op = SystemHintOp SEV;
  when '0000 101' op = SystemHintOp SEVL;
  when '0000 111'
    SEE "XPACLR1";
  when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp TSB;
  when '0010 100'
    op = SystemHintOp CSDB;
  when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0'
    op = otherwise SystemHintOp BTI;
    BTypeCompatible = BTypeCompatible BTI(op2<2:1>);
  otherwise EndOfInstruction(); // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

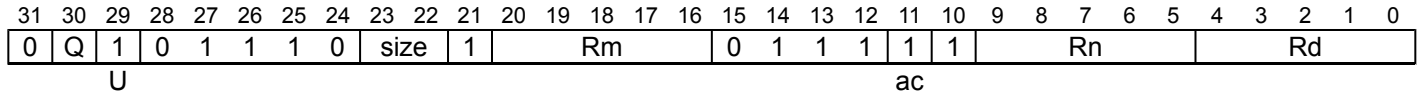
htmldiff from-

(new)

## UABA

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

UABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABAL instruction extracts each source vector from the lower half of each source register, while the UABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	1	0	1	1	1	0	size	1	Rm						0	1	0	1	0	0	Rn						Rd					
U								op																									

### Three registers, not all the same type

```
UABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED



<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

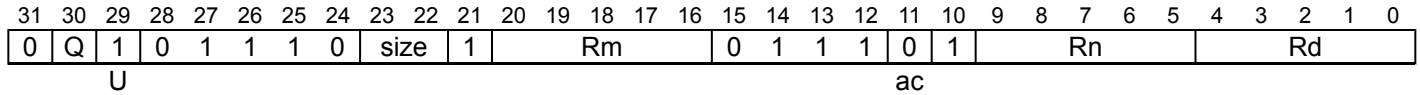
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## UABD

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

UABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## UABDL, UABDL2

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABDL instruction extracts each source vector from the lower half of each source register, while the UABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1				Rm			0	1	1	1	0	0				Rn				Rd		
U								op																							

### Three registers, not all the same type

```
UABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1 - element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:16+09:0445

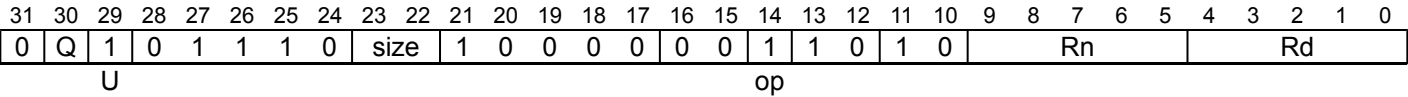
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

UADALP

Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Vector

```
UADALP <Vd>.<Ta>, <Vn>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Vd>Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Ta>Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

- <Vn>Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <Tb>Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## UADDL, UADDL2

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UADDL instruction extracts each source vector from the lower half of each source register, while the UADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	1	0	1	1	1	0	size	1	Rm						0	0	0	0	0	0	Rn						Rd							
U										o1																									

### Three registers, not all the same type

```
UADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED



<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

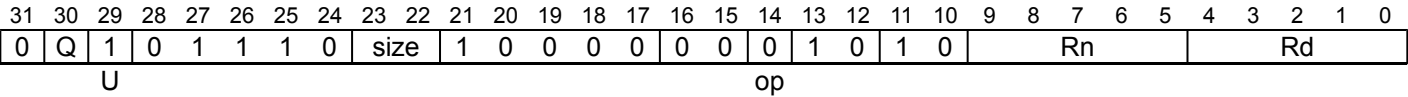
htmldiff from-

(new)

UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Vector

```
UADDLP <Vd>.<Ta>, <Vn>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2*esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1 + op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

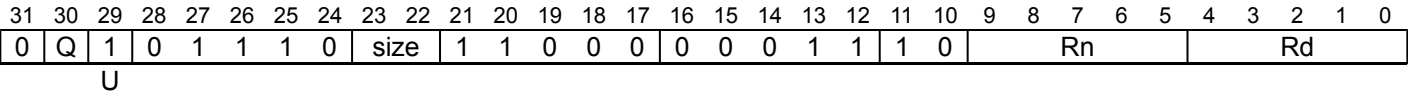
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# UADDLV

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## Advanced SIMD

```
UADDLV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

## Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	H
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d] = sum<2*esize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:18+00:002018-06-16T09:04:45

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## UADDW, UADDW2

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The UADDW instruction extracts vector elements from the lower half of the second source register, while the UADDW2 instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	1	0	1	1	1	0	size	1	Rm						0	0	0	1	0	0	Rn						Rd							
U										o1																									

### Three registers, not all the same type

```
UADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UBFM

Unigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If  $\langle\text{imms}\rangle$  is greater than or equal to  $\langle\text{immr}\rangle$ , this copies a bitfield of  $(\langle\text{imms}\rangle - \langle\text{immr}\rangle + 1)$  bits starting from bit position  $\langle\text{immr}\rangle$  in the source register to the least significant bits of the destination register.

If  $\langle\text{imms}\rangle$  is less than  $\langle\text{immr}\rangle$ , this copies a bitfield of  $(\langle\text{imms}\rangle + 1)$  bits from the least significant bits of the source register to bit position  $(\text{regsize} - \langle\text{immr}\rangle)$  of the destination register, where  $\text{regsize}$  is the destination register size of 32 or 64 bits.

In both cases the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases [LSL \(immediate\)](#), [LSR \(immediate\)](#), [UBFIZ](#), [UBFX](#), [UXTB](#), and [UXTH](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
sf	1	0	1	0	0	1	1	0	N	immr						imms						Rn						Rd					
opc																																	

### 32-bit (sf == 0 && N == 0)

UBFM  $\langle\text{Wd}\rangle$ ,  $\langle\text{Wn}\rangle$ ,  $\# \langle\text{immr}\rangle$ ,  $\# \langle\text{imms}\rangle$

### 64-bit (sf == 1 && N == 1)

UBFM  $\langle\text{Xd}\rangle$ ,  $\langle\text{Xn}\rangle$ ,  $\# \langle\text{immr}\rangle$ ,  $\# \langle\text{imms}\rangle$

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

case opc of
  when '00' inzero = TRUE;  extend = TRUE;    // SBFM
  when '01' inzero = FALSE; extend = FALSE;   // BFM
  when '10' inzero = TRUE;  extend = FALSE;   // UBFM
  when '11' UNDEFINED;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

R = when '11' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

## Assembler Symbols

$\langle\text{Wd}\rangle$	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
$\langle\text{Wn}\rangle$	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
$\langle\text{Xd}\rangle$	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
$\langle\text{Xn}\rangle$	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
$\langle\text{immr}\rangle$	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.



For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

<imms> For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

### Alias Conditions

Alias	Of variant	Is preferred when
<a href="#">LSL (immediate)</a>	32-bit	<code>imms != '011111' &amp;&amp; imms + 1 == immr</code>
<a href="#">LSL (immediate)</a>	64-bit	<code>imms != '111111' &amp;&amp; imms + 1 == immr</code>
<a href="#">LSR (immediate)</a>	32-bit	<code>imms == '011111'</code>
<a href="#">LSR (immediate)</a>	64-bit	<code>imms == '111111'</code>
<a href="#">UBFIZ</a>		<code>UInt(imms) &lt; UInt(immr)</code>
<a href="#">UBFX</a>		<code>BFXPreferred(sf, opc&lt;1&gt;, imms, immr)</code>
<a href="#">UXTB</a>		<code>immr == '000000' &amp;&amp; imms == '000111'</code>
<a href="#">UXTH</a>		<code>immr == '000000' &amp;&amp; imms == '001111'</code>

### Operation

```
bits(datasize) dst = if inzero then Zeros\(\) else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa [v30.25-v29.05](#), AdvSIMD [v27.01-v26.0](#), pseudocode [v85-xml-00bet8\\_rc3-v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UBFX

Unsigned Bitfield Extract copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This is an alias of [UBFM](#). This means:

- The encodings in this description are named to match the encodings of [UBFM](#).
- The description of [UBFM](#) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	0	0	1	1	0	N	immr						imms						Rn				Rd					
opc																															

### 32-bit (sf == 0 && N == 0)

UBFX `<Wd>`, `<Wn>`, `#<lsb>`, `#<width>`

is equivalent to

UBFM `<Wd>`, `<Wn>`, `#<lsb>`, `#(<lsb>+<width>-1)`

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

### 64-bit (sf == 1 && N == 1)

UBFX `<Xd>`, `<Xn>`, `#<lsb>`, `#<width>`

is equivalent to

UBFM `<Xd>`, `<Xn>`, `#<lsb>`, `#(<lsb>+<width>-1)`

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

## Assembler Symbols

<code>&lt;Wd&gt;</code>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<code>&lt;Wn&gt;</code>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<code>&lt;Xd&gt;</code>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<code>&lt;Xn&gt;</code>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<code>&lt;lsb&gt;</code>	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<code>&lt;width&gt;</code>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32- <code>&lt;lsb&gt;</code> . For the 64-bit variant: is the width of the bitfield, in the range 1 to 64- <code>&lt;lsb&gt;</code> .

## Operation

The description of [UBFM](#) gives the operational pseudocode for this instruction.

## Operational information

If `PSTATE.DIT` is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UCVTF (vector, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn				Rd					
U									immh																						

### Scalar

UCVTF <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - (()) then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn				Rd					
U									immb																						

Vector

UCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - (()) then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## UCVTF (vector, integer)

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

### Scalar half precision

UCVTF <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

### Scalar single-precision and double-precision

UCVTF <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

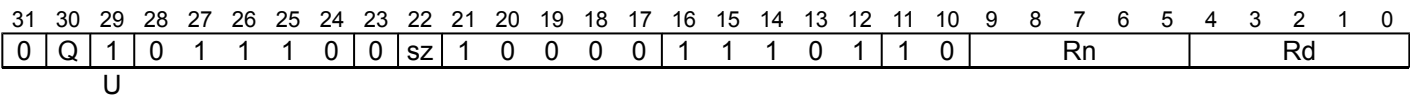
Vector half precision

```
UCVTF <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;
integer d = () then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



Vector single-precision and double-precision

```
UCVTF <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.



Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
FPRounding rounding = FPRoundingMode(FPCR);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

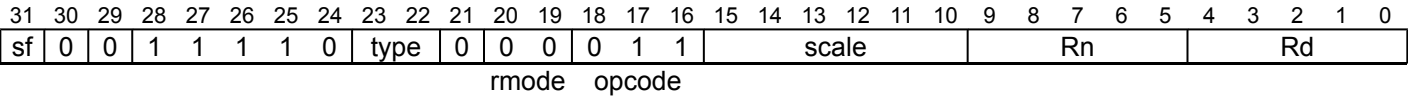
(old)	htmldiff from-	(new)
-------	----------------	-------

UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.



**32-bit to half-precision (sf == 0 && type == 11)**  
(ARMv8.2)

UCVTF <Hd>, <Wn>, #<fbits>

**32-bit to single-precision (sf == 0 && type == 00)**

UCVTF <Sd>, <Wn>, #<fbits>

**32-bit to double-precision (sf == 0 && type == 01)**

UCVTF <Dd>, <Wn>, #<fbits>

**64-bit to half-precision (sf == 1 && type == 11)**  
(ARMv8.2)

UCVTF <Hd>, <Xn>, #<fbits>

**64-bit to single-precision (sf == 1 && type == 00)**

UCVTF <Sd>, <Xn>, #<fbits>

**64-bit to double-precision (sf == 1 && type == 01)**

UCVTF <Dd>, <Xn>, #<fbits>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;

case type of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if when '10' UnallocatedEncoding();
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - else UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

case opcode<2:1>:rmode of
  when '00 11' // FCVTZ
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  otherwise
    UnallocatedEncoding;
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	<p>For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".</p> <p>For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".</p>

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding);
    V[d] = fltval;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

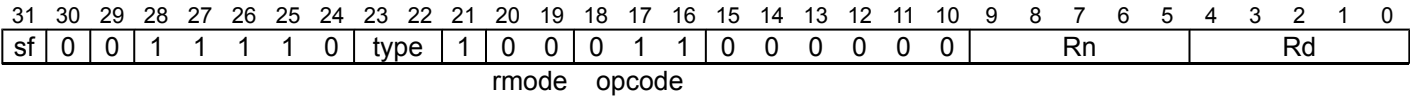
(old)	htmldiff from-	(new)
-------	----------------	-------

UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



**32-bit to half-precision (sf == 0 && type == 11)**  
(ARMv8.2)

UCVTF <Hd>, <Wn>

**32-bit to single-precision (sf == 0 && type == 00)**

UCVTF <Sd>, <Wn>

**32-bit to double-precision (sf == 0 && type == 01)**

UCVTF <Dd>, <Wn>

**64-bit to half-precision (sf == 1 && type == 11)**  
(ARMv8.2)

UCVTF <Hd>, <Xn>

**64-bit to single-precision (sf == 1 && type == 00)**

UCVTF <Sd>, <Xn>

**64-bit to double-precision (sf == 1 && type == 01)**

UCVTF <Dd>, <Xn>

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = else UnallocatedEncoding();

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR);
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI;
  when '11 00' // FMOV
    if fltsize != 16 && fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
    op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    part = 1;
    fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = () then UnallocatedEncoding();
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  otherwise
    UnallocatedEncoding();
  otherwise
    UNDEFINED; ();

```

## Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.



<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

## UDF

Permanently Undefined generates an Undefined Instruction exception (ESR\_ELx.EC = 0b0000000). The encodings for UDF used in this section are defined as permanently UNDEFINED in the ARMv8-A architecture.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	imm16															

### Integer

```
UDF #<imm>
```

```
// The imm16 field is ignored by hardware.
UNDEFINED;
```

### Assembler Symbols

<imm> is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. The PE ignores the value of this constant.

### Operation

```
// No operation.
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UDOT (by element)

Dot Product unsigned arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

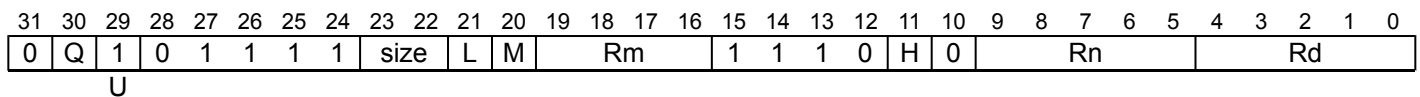
Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

[ID\\_AA64ISAR0\\_EL1](#).DP indicates whether this instruction is supported.

### Vector

(ARMv8.2)



### Vector

UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm> .4B[<index>]

```

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U=='0');

integer d = if size != '10' then ReservedValue();
boolean signed = (U=='0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the element index, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## UDOT (vector)

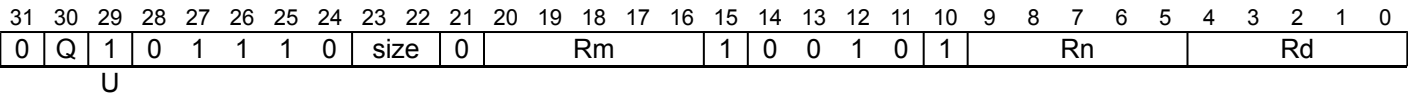
Dot Product unsigned arithmetic (vector). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the `CPACR_ELI`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In ARMv8.2 and ARMv8.3, this is an OPTIONAL instruction. From ARMv8.4 it is mandatory for all implementations to support it.

`ID_AA64ISAR0_ELI`.DP indicates whether this instruction is supported.

### Three registers of the same type (ARMv8.2)



### Three registers of the same type

UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveDOTPExt() then UNDEFINED;
if size!= '10' then UNDEFINED;
boolean signed = (U=='0');
integer d =if size!= '10' then ReservedValue();
boolean signed = (U=='0');
integer d =UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S
- <Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B
- <Vm>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

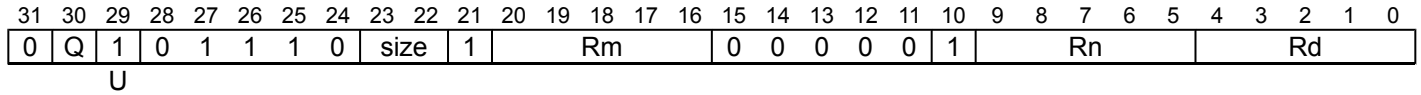
(new)

## UHADD

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see [URHADD](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

UHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

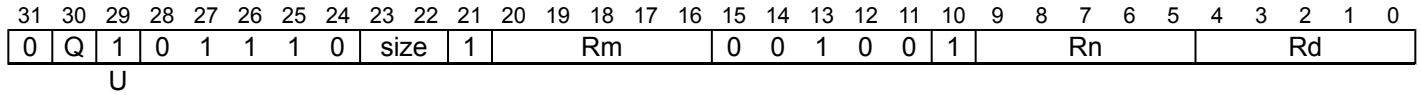
(old)	htmldiff from-	(new)
-------	----------------	-------



## UHSUB

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

```
UHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

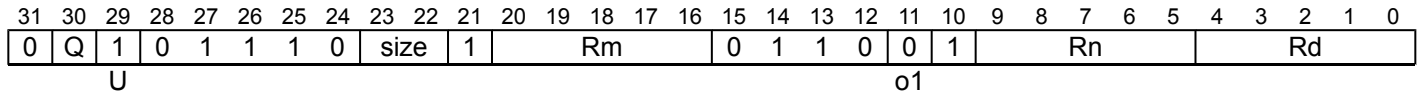
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## UMAX

Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

UMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

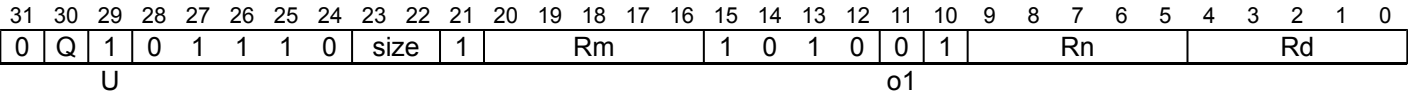
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# UMAXP

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## Three registers of the same type

```
UMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler Symbols

- <Vd>Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn>Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

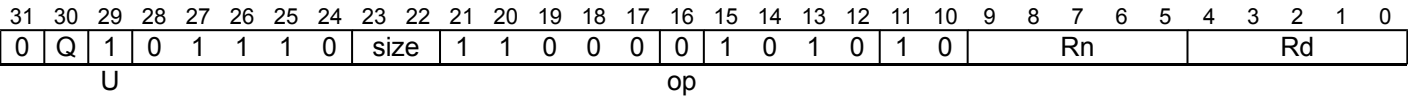
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

UMAXV

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Advanced SIMD

```
UMAXV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

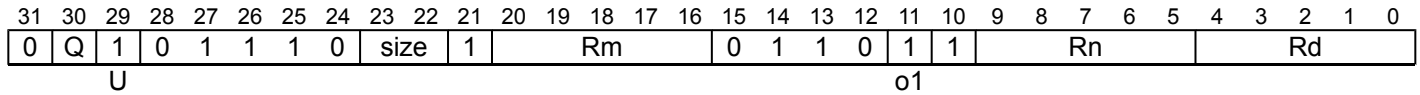
(old)	htmldiff from-	(new)
-------	----------------	-------



## UMIN

Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

```
UMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

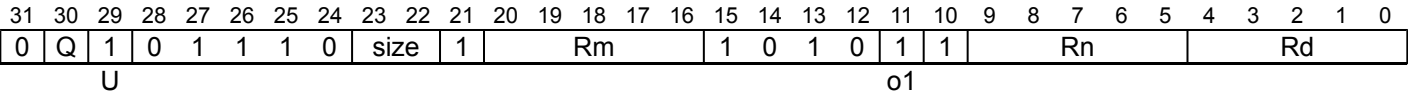
Internal version only: isa ~~v30.25~~v29.05, AdvSIMD ~~v27.01~~v26.0, pseudocode ~~v85-xml-00bet8\_rc3~~v35.3 ; Build timestamp: ~~2018-09-13T13:20:16+09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# UMINP

Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## Three registers of the same type

```
UMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

## Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

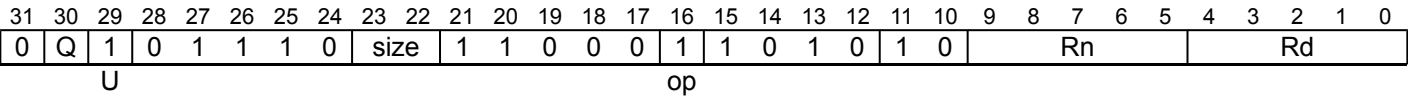
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

# UMINV

Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## Advanced SIMD

```
UMINV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

## Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## UMLAL, UMLAL2 (by element)

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register, while the UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size	L	M	Rm				0	0	1	0	H	0	Rn				Rd						
U										o2																					

### Vector

UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```

integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UNDEFINED;

integer d = otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(idxdsize)  operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2) < 2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25v29.05**, AdvSIMD **v27.01v26.0**, pseudocode **v85-xml-00bet8\_rc3v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register, while the UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	1	0	1	1	1	0	size		1	Rm						1	0	0	0	0	0	Rn						Rd						
U										o1																									

### Three registers, not all the same type

```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25-v29.05**, AdvSIMD **v27.01-v26.0**, pseudocode **v85-xml-00bet8\_rc3-v35.3**; Build timestamp: **2018-09-13T13:2018-06-16T09:0445**

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLSL instruction extracts vector elements from the lower half of the first source register, while the UMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size	L	M			Rm		0	1	1	0	H	0					Rn					Rd	
U										o2																					

### Vector

UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```

integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise UNDEFINED;
integer d = otherwise UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(idxdsize)  operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2) < 2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMLSL instruction extracts each source vector from the lower half of each source register, while the UMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	Rm						1	0	1	0	0	0	Rn						Rd			
U										o1																					

### Three registers, not all the same type

```
UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25-v29.05~~, AdvSIMD ~~v27.01-v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3-v35.3~~; Build timestamp: ~~2018-09-13T13:20:18-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

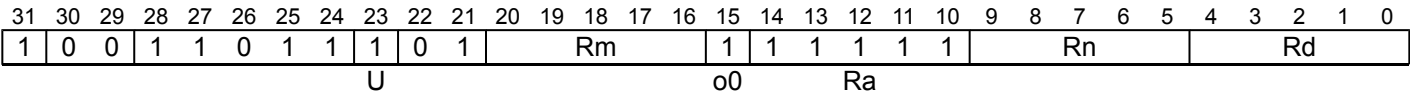
(old)	htmldiff from-	(new)
-------	----------------	-------

UMNEGL

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of [UMSUBL](#). This means:

- The encodings in this description are named to match the encodings of [UMSUBL](#).
- The description of [UMSUBL](#) gives the operational pseudocode for this instruction.



64-bit

UMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

[UMSUBL](#) <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

- <Xd>
- Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>
- Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm>
- Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of [UMSUBL](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMOV

Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [MOV \(to general\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	imm5					0	0	1	1	1	1	Rn					Rd				

### 32-bit (Q == 0)

UMOV <Wd>, <Vn>.<Ts>[<index>]

### 64-reg,UMOV-64-reg (Q == 1 && imm5 == x1000)

UMOV <Xd>, <Vn>.<Ts>[<index>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when '0xxx1' size = 0;      // UMOV Wd, Vn.B
    when '0xxx10' size = 1;     // UMOV Wd, Vn.H
    when '0xx100' size = 2;     // UMOV Wd, Vn.S
    when '1x1000' size = 3;     // UMOV Xd, Vn.D
    otherwise UNDEFINED;

integer idxsize = if imm5<4> == '1' then 128 else 64;
integer index = otherwise UnallocatedEncoding();

integer idxsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4>:size+1);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <Ts> For the 32-bit variant: is an element size specifier, encoded in "imm5":

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

For the 64-reg,UMOV-64-reg variant: is an element size specifier, encoded in "imm5":

imm5	<Ts>
x0000	RESERVED
xxxx1	RESERVED
xxx10	RESERVED
xx100	RESERVED
x1000	D

- <index> For the 32-bit variant: is the element index encoded in "imm5":



imm5	<index>
xx000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>

For the 64-reg,UMOV-64-reg variant: is the element index encoded in "imm5<4>".

## Alias Conditions

Alias	Is preferred when
<a href="#">MOV (to general)</a>	imm5 == 'x1000'
<a href="#">MOV (to general)</a>	imm5 == 'xx100'

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];

X[d] = ZeroExtend(Elem[operand, index, esize], datasize);

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMULL instruction extracts vector elements from the lower half of the first source register, while the UMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTL\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size	L	M					Rm	1	0	1	0	H	0										Rd
U																															

### Vector

```
UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L);   Rmhi = M;
  otherwise UNDEFINED;

integer d = otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H : L : M
10	H : L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)   operand1 = Vpart[n, part];
bits(idxdsize)   operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UMULL, UMULL2 (vector)

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMULL instruction extracts each source vector from the lower half of each source register, while the UMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	Rm						1	1	0	0	0	0	Rn						Rd			
U																															

### Three registers, not all the same type

```
UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)    operand1 = Vpart[n, part];
bits(datasize)    operand2 = Vpart[m, part];
bits(2*datasize)  result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1 * element2)<2*esize-1:0>;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

# UQADD

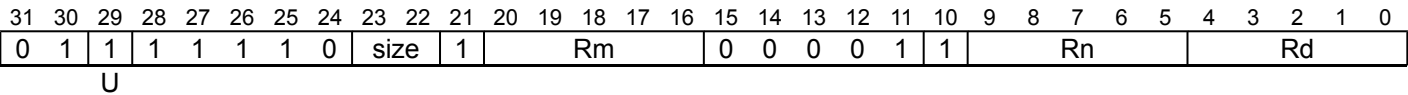
Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit `FPSR.QC` is set.

Depending on the settings in the `CPACR_ELI`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

## Scalar

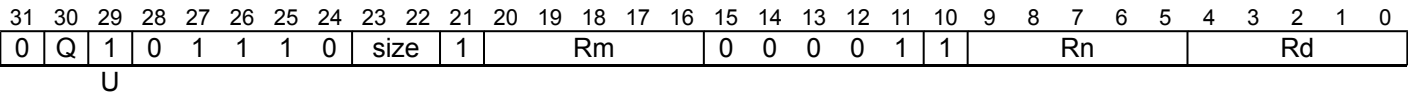


## Scalar

```
UQADD <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

## Vector



## Vector

```
UQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

## Assembler Symbols

- <V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D
- <d>

Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;

```

## UQRSHL

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see [UQSHL](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	1	1	0	size	1	Rm						0	1	0	1	1	1	Rn						Rd							
U								R S																											

### Scalar

UQRSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED; if S == '0' && size != '11' then ReservedValue();
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	1	0	1	1	1	0	size	1	Rm						0	1	0	1	1	1	Rn						Rd							
U										R S																									

### Vector

UQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler Symbols

<V> Is a width specifier, encoded in “size”:



size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UQRSHRN, UQRSHRN2

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [UQSHRN](#).

The UQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	0	0	1	1	1	Rn				Rd					
U									immh				op																		

### Scalar

UQRSHRN <Vb><d>, <Va><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	1	1	Rn				Rd					
U									immh				op																		

Vector

UQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UQSHL (immediate)

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see [UQSRHL](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	1	1	1	0	1	Rn				Rd					
U									immh				op																		

### Scalar

UQSHL <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << if immh == '0000' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UnallocatedEncoding(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	1	1	0	1	Rn				Rd					
U									immh				op																		

Vector

UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UnallocatedEncoding(immh:immb) - esize;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt (immh:immb)-8)
001x	(UInt (immh:immb)-16)
01xx	(UInt (immh:immb)-32)
1xxx	(UInt (immh:immb)-64)

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13](#)~~2018-06-16T09~~:[0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UQSHL (register)

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see [UQRSHL](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1				Rm			0	1	0	0	1	1										
U								R S																							

### Scalar

UQSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED; if S == '0' && size != '11' then ReservedValue();
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1				Rm			0	1	0	0	1	1										
U								R S																							

### Vector

UQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:



size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UQSHRN, UQSHRN2

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [UQSHRN](#).

The UQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn				Rd					
U									immh				op																		

### Scalar

UQSHRN [<Vb><d>](#), [<Va><n>](#), #[<shift>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh == '0000' then ReservedValue\(\);
if immh<3> == '1' then ReservedValue\(\);
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn					Rd				
U									immh									op													

Vector

UQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa [v30.25-v29.05](#), AdvSIMD [v27.01-v26.0](#), pseudocode [v85-xml-00bet8\\_rc3-v35.3](#); Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

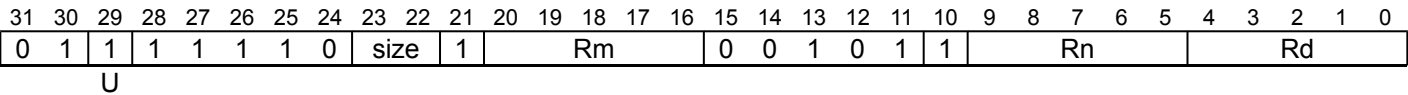
(new)

UQSUB

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar

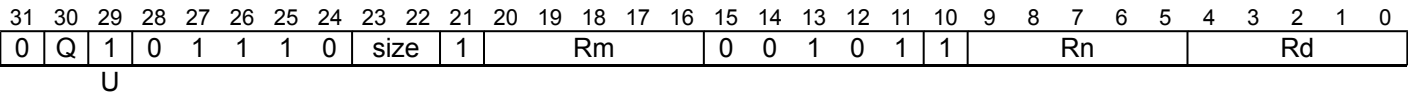


Scalar

```
UQSUB <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector



Vector

```
UQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V>	Is a width specifier, encoded in “size”:										
	<table><tr><th>size</th><th>&lt;V&gt;</th></tr><tr><td>00</td><td>B</td></tr><tr><td>01</td><td>H</td></tr><tr><td>10</td><td>S</td></tr><tr><td>11</td><td>D</td></tr></table>	size	<V>	00	B	01	H	10	S	11	D
size	<V>										
00	B										
01	H										
10	S										
11	D										
<d>	Is the number of the SIMD&FP destination register, in the "Rd" field.										
<n>	Is the number of the first SIMD&FP source register, encoded in the "Rn" field.										

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<b>&lt;Vm&gt;</b>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
-------------------	---

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8 rc3v35.3; Build timestamp: 2018-09-13T13:20:09.0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

```
htmldiff from-
```

(new)

## UQXTN, UQXTN2

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The UQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQXTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	size				1	0	0	0	0	1	0	1	0	0	1	0	Rn				Rd				
U																															

### Scalar

UQXTN <Vb><d>, <Va><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size				1	0	0	0	0	1	0	1	0	0	1	0	Rn				Rd			
U																															

### Vector

UQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```



URECPE

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn				Rd					

Vector

```
URECPE <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; if sz == '1' then ReservedValue();
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

- <Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRecipEstimate(element);

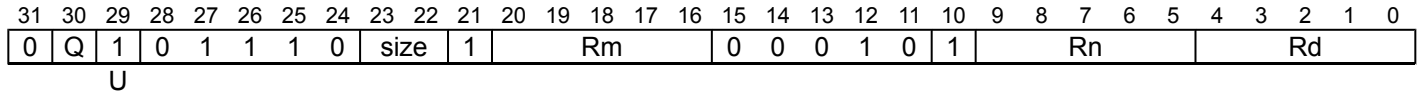
V[d] = result;
```

## URHADD

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see [UHADD](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Three registers of the same type

URHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1 + element2 + 1) < esize:1 >;

V[d] = result;
```

(old)

htmldiff from-

(new)

## URSHL

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1	Rm					0	1	0	1	0	1	Rn					Rd					
U										R										S											

### Scalar

URSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED; if S == '0' && size != '11' then ReservedValue();
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	Rm					0	1	0	1	0	1	Rn					Rd					
U										R										S											

### Vector

URSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## URSHR

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [USHR](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	0	1	0	0	1	Rn				Rd					
U									immh				o1 o0																		

### Scalar

URSHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) -if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	0	1	0	0	1	Rn				Rd					
U									immh							o1		o0													

Vector

```
URSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



URSQRTE

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn					Rd				

Vector

```
URSQRTE <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; if sz == '1' then ReservedValue();
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

- <Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRSqrtEstimate(element);

V[d] = result;
```

## URSRA

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [USRA](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	0	1	1	0	1	Rn				Rd					
U									immh				o1 o0																		

### Scalar

URSRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) -if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	0	1	1	0	1	Rn				Rd					
U									immh				o1			o0															

Vector

URSRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## USHL

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see [URSHL](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	1	1	1	1	0	size	1	Rm						0	1	0	0	0	1	Rn						Rd				
U										R S																						

### Scalar

USHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED; if S == '0' && size != '11' then ReservedValue();
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	Q	1	0	1	1	1	0	size	1	Rm						0	1	0	0	0	1	Rn						Rd				
U										R S																						

### Vector

USHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## USHLL, USHLL2

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The USHLL instruction extracts vector elements from the lower half of the source register, while the USHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [UXTL](#), [UXTL2](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	1	0	0	1	Rn				Rd					
U									immh																						

### Vector

```
USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt (immh:immb) - 8)
001x	(UInt (immh:immb) - 16)
01xx	(UInt (immh:immb) - 32)
1xxx	RESERVED

## Alias Conditions

Alias	Is preferred when
<a href="#">UXTL, UXTL2</a>	immb == '000' && <a href="#">BitCount</a> (immh) == 1

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa [v30.25](#)[v29.05](#), AdvSIMD [v27.01](#)[v26.0](#), pseudocode [v85-xml-00](#)[bet8](#) [rc3](#)[v35.3](#); Build timestamp: [2018-09-13](#)[T13](#)[2018-06-16](#)[T09](#)[0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## USHR

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [URSHR](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	0	0	0	0	1	Rn				Rd					
U									immh				o1 o0																		

### Scalar

USHR [<V><d>](#), [<V><n>](#), #[<shift>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) -if immh<3> != '1' then ReservedValue\(\);
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	0	0	0	0	1	Rn				Rd					
U									immh				o1			o0															

Vector

USHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## USQADD

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1	0	0	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
U																															

### Scalar

```
USQADD <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	0	0	1	1	1	0	Rn				Rd					
U																															

### Vector

```
USQADD <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

## Assembler Symbols

<V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;

```

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## USRA

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [URSR4](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	0	0	1	0	1	Rn				Rd					
U									immh				o1		o0																

### Scalar

USRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) -if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	0	0	1	0	1	Rn				Rd					
U									immb				o1		o0																

Vector

USRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## USUBL, USUBL2

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The USUBL instruction extracts each source vector from the lower half of each source register, while the USUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	Q	1	0	1	1	1	0	size	1	Rm						0	0	1	0	0	0	Rn						Rd							
U										o1																									

### Three registers, not all the same type

```
USUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa **v30.25**~~v29.05~~, AdvSIMD **v27.01**~~v26.0~~, pseudocode **v85-xml-00bet8\_rc3**~~v35.3~~; Build timestamp: **2018-09-13T13:20:04**~~2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## USUBW, USUBW2

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The USUBW instruction extracts vector elements from the lower half of the first source register, while the USUBW2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	Rm						0	0	1	1	0	0	Rn						Rd			
U										o1																					

### Three registers, not all the same type

```
USUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa ~~v30.25~~~~v29.05~~, AdvSIMD ~~v27.01~~~~v26.0~~, pseudocode ~~v85-xml-00bet8\_rc3~~~~v35.3~~; Build timestamp: ~~2018-09-13T13:2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

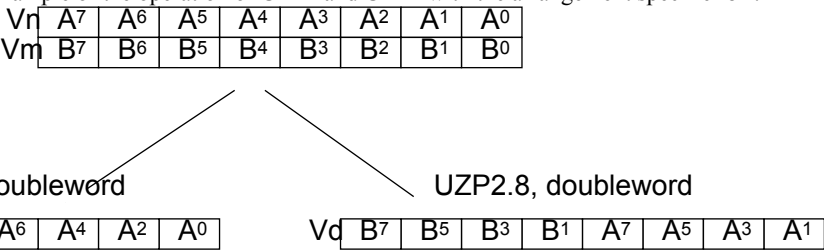
(new)

UZP1

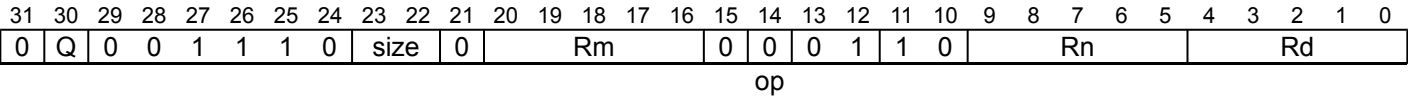
Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can be used with UZP2 to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Advanced SIMD

```
UZP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;
integer e;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

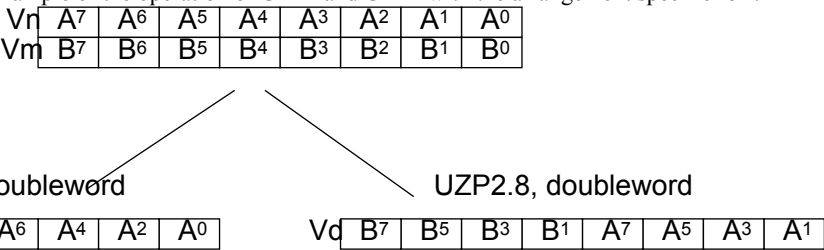
(old)	htmldiff from-	(new)
-------	----------------	-------

UZP2

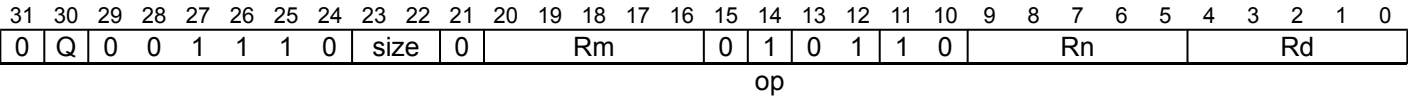
Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can be used with UZP1 to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Advanced SIMD

```
UZP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;
integer e;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



## WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event*.

As described in *Wait For Event mechanism and Send event*, the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions.*
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions.*
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions.*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0	1	1	1	1	1
																CRm								op2							

## System

WFE

[SystemHintOp](#) op;

case CRm:op2 of

```

when '0000 000' op = SystemHintOp\_NOP;
when '0000 001' op = SystemHintOp\_YIELD;
when '0000 010' op = SystemHintOp\_WFE;
when '0000 011' op = SystemHintOp\_WFI;
when '0000 100' op = SystemHintOp\_SEV;
when '0000 101' op = SystemHintOp\_SEVL;
when '0000 111'
    SEE "XPACLRI";
when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
when '0010 000'
    if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp\_ESB;
when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp\_PSB;
when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp\_TSB;
when '0010 100'
    op = SystemHintOp\_CSDB;
when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
when '0100 xx0'
    op = otherwise SystemHintOp\_BTI;
    BTypeCompatible = BTypeCompatible\_BTI(op2<2:1>);
otherwise EndOfInstruction(); // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see [Wait For Interrupt](#).

As described in [Wait For Interrupt](#), the execution of a WFI instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- [Traps to EL1 of EL0 execution of WFE and WFI instructions](#).
- [Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions](#).
- [Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1
																CRm				op2											

## System

WFI

```

SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 111'
    SEE "XPACLR1";
  when '0001 xxx'
    SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 100'
    op = SystemHintOp_CSDB;
  when '0011 xxx'
    SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";
  when '0100 xx0'
    op = otherwise SystemHintOp_BTI;
    BTypeCompatible = BTypeCompatible_BTI(op2<2:1>);
  otherwise EndOfInstruction(); // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWfxTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWfxTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWfxTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## XAFlag

Convert floating-point condition flags from external format to ARM format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from an alternative representation required by some software to a form representing the result of an ARM floating-point scalar compare instruction.

### System (ARMv8.5)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	(0)	(0)	(0)	(0)	0	0	1	1	1	1	1	1

CRm

### System

XAFlag

```
if !HaveFlagFormatExt() then UNDEFINED;
```

### Operation

```
bit N = NOT(PSTATE.C) AND NOT(PSTATE.Z);
bit Z = PSTATE.Z AND PSTATE.C;
bit C = PSTATE.C OR PSTATE.Z;
bit V = NOT(PSTATE.C) AND PSTATE.Z;
```

```
PSTATE.N = N;
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = V;
```

Internal version only: isa v30.25, AdvSIMD v27.01, pseudocode v85-xml-00bet8\_rc3 ; Build timestamp: 2018-09-13T13:04

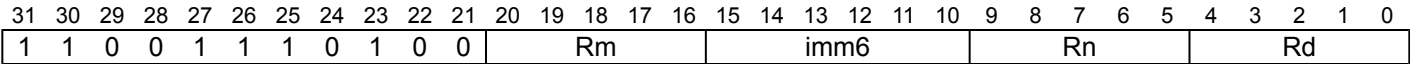
Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# XAR

Exclusive OR and Rotate performs a bitwise exclusive OR of the 128-bit vectors in the two source SIMD&FP registers, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SHA* is implemented.

## Advanced SIMD (ARMv8.2)



## Advanced SIMD

```
XAR <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>
```

```
if !HaveSHA3Ext() then UNDEFINED;
integer d =() then UnallocatedEncoding();
integer d = UInt (Rd) ;
integer n = UInt (Rn) ;
integer m = UInt (Rm) ;
```

## Assembler Symbols

- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- <imm6>

Is a rotation right, encoded in "imm6".

## Operation

```
AArch64.CheckFPAdvSIMDEnabled() ;

bits(128) Vm = V[m] ;
bits(128) Vn = V[n] ;
bits(128) tmp;
tmp = Vn EOR Vm;
V[d] = ROR (tmp<127:64>, UInt (imm6)) :ROR (tmp<63:0>, UInt (imm6)) ;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

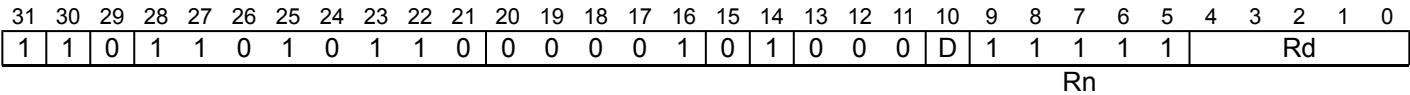
XPACD, XPACI, XPACLRI

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for XPACI and XPACD, and is in LR for XPACLRI.

The XPACD instruction is used for data addresses, and XPACI and XPACLRI are used for instruction addresses.

It has encodings from 2 classes: [Integer](#) and [System](#)

Integer  
(ARMv8.3)



XPACD (D == 1)

```
XPACD <Xd>
```

XPACI (D == 0)

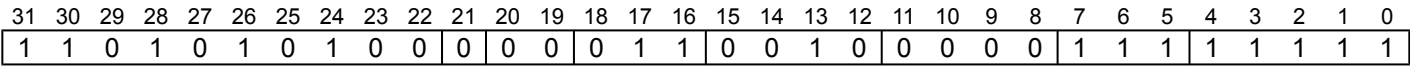
```
XPACI <Xd>
```

```
boolean data = (D == '1');
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if n != 31 then UNDEFINED; () then UnallocatedEncoding();
if n != 31 then UnallocatedEncoding();
```

System  
(ARMv8.3)



System

```
XPACLRI

integer d = 30;
boolean data = FALSE;
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

```
if HavePACExt() then
    X[d] = Strip(X[d], data);
```

(old)

htmldiff from-

(new)



## XTN, XTN2

Extract Narrow. This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

The XTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the XTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	1	0	0	1	0	1	0	Rn				Rd					

### Vector

XTN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3 ; Build timestamp: 2018-09-13T13:20:18-06:16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

## YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see [The YIELD instruction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1
																CRm				op2											

### System

YIELD

[SystemHintOp](#) op;

case CRm:op2 of

```
when '0000 000' op = SystemHintOp NOP;
when '0000 001' op = SystemHintOp YIELD;
when '0000 010' op = SystemHintOp WFE;
when '0000 011' op = SystemHintOp WFI;
when '0000 100' op = SystemHintOp SEV;
when '0000 101' op = SystemHintOp SEVL;
when '0000 111'
```

SEE "XPACLR1";

```
when '0001 xxx'
```

SEE "PACIA1716, PACIB1716, AUTIA1716, AUTIB1716";

```
when '0010 000'
```

```
if !HaveRASExt() then EndOfInstruction();
```

// Instruction executes as NOP

```
op = SystemHintOp ESB;
```

```
when '0010 001'
```

```
if !HaveStatisticalProfiling() then EndOfInstruction();
```

// Instruction executes as NOP

```
op = SystemHintOp PSB;
```

```
when '0010 010'
```

```
if !HaveSelfHostedTrace() then EndOfInstruction();
```

// Instruction executes as NOP

```
op = SystemHintOp TSB;
```

```
when '0010 100'
```

```
op = SystemHintOp CSDB;
```

```
when '0011 xxx'
```

SEE "PACIAZ, PACIASP, PACIBZ, PACIBSP, AUTIAZ, AUTIASP, AUTIBZ, AUTIBSP";

```
when '0100 xx0'
```

```
op = otherwise SystemHintOp BTI;
```

```
BTypeCompatible = BTypeCompatible BTI(op2<2:1>);
```

```
otherwise EndOfInstruction();
```

// Instruction executes as NOP

## Operation

```
case op of
  when SystemHintOp\_YIELDHint\_Yield\(\);

  when SystemHintOp\_WFE
    if IsEventRegisterSet\(\) then
      ClearEventRegister\(\);
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, TRUE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, TRUE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, TRUE\);
      WaitForEvent\(\);

  when SystemHintOp\_WFI
    if !InterruptPending\(\) then
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFXTrap\(EL1, FALSE\);
      if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && !IsInHost\(\) then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFXTrap\(EL2, FALSE\);
      if HaveEL\(EL3\) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFXTrap\(EL3, FALSE\);
      WaitForInterrupt\(\);

  when SystemHintOp\_SEVSendEvent\(\);

  when SystemHintOp\_SEVLSendEventLocal\(\);

  when SystemHintOp\_ESBSynchronizeErrors\(\);
    AArch64.ESBOperation\(\);
    if EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} then AArch64.vESBOperation\(\);
    TakeUnmaskedErrorInterrupts\(\);

  when SystemHintOp\_PSBProfilingSynchronizationBarrier\(\);

  when SystemHintOp\_TSB
    TraceSynchronizationBarrier\(\);

  when SystemHintOp\_CSDBConsumptionOfSpeculativeDataBarrier\(\);

  when otherwise // do nothing SystemHintOp\_BTI
    BTypeNext = '00';

  otherwise // do nothing
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

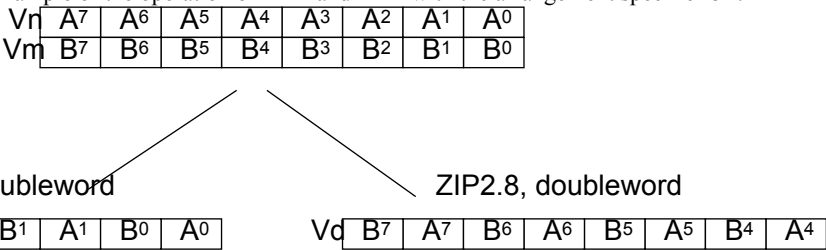
(new)

ZIP1

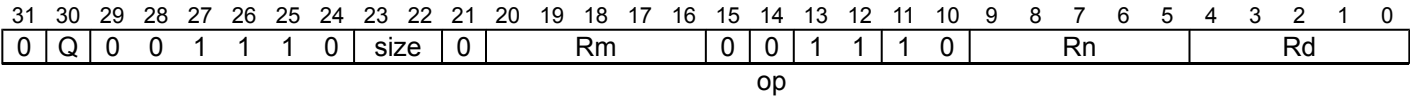
Zip vectors (primary). This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

This instruction can be used with ZIP2 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Advanced SIMD

ZIP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;
integer p;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

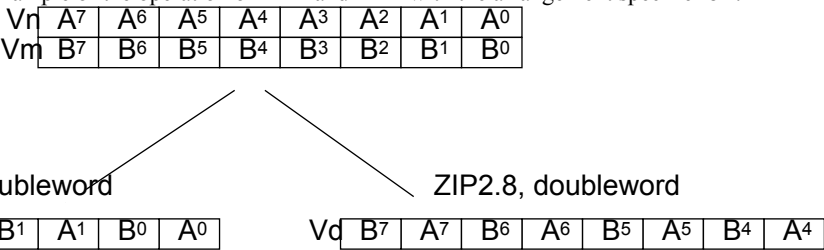
(new)

ZIP2

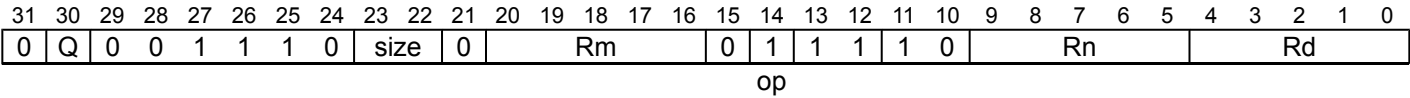
Zip vectors (secondary). This instruction reads adjacent vector elements from the lower half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

This instruction can be used with ZIP1 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Advanced SIMD

ZIP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;
integer p;

for p = 0 to pairs-1
  Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
  Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v30.25v29.05, AdvSIMD v27.01v26.0, pseudocode v85-xml-00bet8\_rc3v35.3; Build timestamp: 2018-09-13T13:2018-06-16T09:0445

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------



(old)

htmldiff from-

(new)

## Top-level encodings for A64

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			op0																												

### Decode fields

op0

### Instruction details

0000	00xx	UNALLOCATED	Reserved
0001	100x	Data Processing -- Immediate	UNALLOCATED
001x	101x	Branches, Exception Generating and System instructions	UNALLOCATED
100x	x1x0	Data Processing -- Immediate	Loads and Stores
101x	x101	Branches, Exception Generating and System instructions	Data Processing -- Register
x1x0	x111	Loads and Stores	Data Processing -- Scalar Floating-Point and Advanced SIMD
x101		Data Processing -- Register	
x111		Data Processing -- Scalar Floating-Point and Advanced SIMD	

## Reserved

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
op0			0000			op1																										

### Decode fields

op0

op1

### Instruction details

000	000000000	UDF
000	!= 000000000	UNALLOCATED
!= 000	000000000	UNALLOCATED

## Data Processing -- Immediate

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			100			op0			op1																						

### Decode fields

op0

op1

### Instruction details

00	00x		PC-rel. addressing
01	01x	1x	Add/subtract (immediate, with tags)Add/subtract (immediate)
01	100	!= 1x	Add/subtract (immediate)Logical (immediate)
10	101	0x	Logical (immediate)Move wide (immediate)
10	110	1x	Move wide (immediate)Bitfield
11	111	0x	BitfieldExtract
	11	1x	Extract

## PC-rel. addressing

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op	immlo				1	0	0	0	0	immhi																		Rd			

Decode fields		Instruction Details	
op			
0		ADR	
1		ADRP	

### Add/subtract (immediate, with tags)

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	0	0	0	1	1	shift	0	imm12	uimm6				op3		uimm4				Rn				Rd					

Decode fields			Instruction Details		Architecture Version	
sf	op	S				
0			UNALLOCATED		!	
1		1	UNALLOCATED		!	
1	0	0	ADDG		ARMv8.5	
1	1	0	SUBG		ARMv8.5	

Decode fields				Instruction Details	
sf	op	S	shift		
			1x	UNALLOCATED	
0	0	0		ADD (immediate) — 32-bit	
0	0	1		ADDS (immediate) — 32-bit	
0	1	0		SUB (immediate) — 32-bit	
0	1	1		SUBS (immediate) — 32-bit	
1	0	0		ADD (immediate) — 64-bit	
1	0	1		ADDS (immediate) — 64-bit	
1	1	0		SUB (immediate) — 64-bit	
1	1	1		SUBS (immediate) — 64-bit	

### Add/subtract Logical (immediate)

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
sf	op	S	1	0	0	0	1	!= 1x										imm12										Rn				Rd			
shift																																			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
sf	opc	1	0	0	1	0	0	N				immr						imms						Rn				Rd							

The following constraints also apply to this encoding: shift<1> != 1x && shift<1> != 1x

Decode fields				Instruction Details	
sf	op	opc	SN		
0	0		01	ADD (immediate) — UNALLOCATED 32-bit	
0	0		1	ADDS (immediate) — 32-bit	
0	00		0	AND (immediate) — 32-bit	
0	101		0	SUB (immediate) ORR (immediate) — 32-bit	
0	110		10	SUBS (immediate) EOR (immediate) — 32-bit	
1	0		0	ADD (immediate) — 64-bit	
0	11		0	ANDS (immediate) — 32-bit	
1	000		1	ADDS (immediate) AND (immediate) — 64-bit	

Decode fields			Instruction Details
sf	op	pe	SN
1	101	0	SUB (immediate)ORR (immediate) — 64-bit
1	110	1	SUBS (immediate)EOR (immediate) — 64-bit
1	11		ANDS (immediate) — 64-bit

**LogicalMove wide (immediate)**

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc	1	0	0	1	0	04	Nhw				immr	imm16						imms						Rn					Rd	

Decode fields			Instruction Details
sf	opc	N	hw
0		1	UNALLOCATED
	01		UNALLOCATED
0	00	01x	AND (immediate) — UNALLOCATED 32-bit
0	0100	0	ORR (immediate)MOVN — 32-bit
0	10	0	EOR (immediate)MOVZ — 32-bit
0	11	0	ANDS (immediate)MOVK — 32-bit
1	00		AND (immediate)MOVN — 64-bit
1	0110		ORR (immediate)MOVZ — 64-bit
1	1011		EOR (immediate)MOVK — 64-bit
1	11		ANDS (immediate) — 64-bit

**Move wide (immediate)Bitfield**

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc	1	0	0	1	0	1	hw	imm16												Rd										
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc	1	0	0	1	1	0	N	immr				imms				Rn				Rd										

Decode fields			Instruction Details
sf	opc	hw	N
	0111		UNALLOCATED
0		1x1	UNALLOCATED
0	00		MOVNSBFM — 32-bit
0	1001		MOVZBFM — 32-bit
0	1110		MOVKUBFM — 32-bit
1	00		MOVN — 64-bit
1		0	UNALLOCATED
1	1000		MOVZSBFM — 64-bit
1	1101		MOVKBFM — 64-bit
1	10	1	UBFM — 64-bit

**BitfieldExtract**

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc	op24	1	0	0	1	1	04	N		e0	Rm	immr						imms					Rn						Rd	

Decode fields			Instruction Details
sf	opc	N	
	11		UNALLOCATED
0		1	UNALLOCATED
0	00	0	<a href="#">SBFM — 32-bit</a>
0	01	0	<a href="#">BFM — 32-bit</a>
0	10	0	<a href="#">UBFM — 32-bit</a>
1		0	UNALLOCATED
1	00	1	<a href="#">SBFM — 64-bit</a>
1	01	1	<a href="#">BFM — 64-bit</a>
1	10	1	<a href="#">UBFM — 64-bit</a>

Decode fields					Instruction Details
sf	op21	N	o0	imms	
	x1				UNALLOCATED
	00		1		UNALLOCATED
	1x				UNALLOCATED
0				1xxxxx	UNALLOCATED
0		1			UNALLOCATED
0	00	0	0	0xxxxx	<a href="#">EXTR — 32-bit</a>
1		0			UNALLOCATED
1	00	1	0		<a href="#">EXTR — 64-bit</a>

Extract

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op21	1	0	0	1	1	1	N	o0	Rm					imms					Rn					Rd						

Decode fields					Instruction Details
sf	op21	N	o0	imms	
	x1				UNALLOCATED
	00		1		UNALLOCATED
	1x				UNALLOCATED
0				1xxxxx	UNALLOCATED
0		1			UNALLOCATED
0	00	0	0	0xxxxx	<a href="#">EXTR — 32-bit</a>
1		0			UNALLOCATED
1	00	1	0		<a href="#">EXTR — 64-bit</a>

Branches, Exception Generating and System instructions

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0			101			op1															op2										

Decode fields		op2	Instruction details
op0	op1		
010	0xxxxxxxxxxxxxxxxx		<a href="#">Conditional branch (immediate)</a>
010	1xxxxxxxxxxxxxxxxx		UNALLOCATED
110	00xxxxxxxxxxxxxxxxx		<a href="#">Exception generation</a>

110	010000000x001x		UNALLOCATED
110	0100000010001x		UNALLOCATED
110	01000000110010	11111	<a href="#">Hints</a>
110	01000000110010	!= 11111	UNALLOCATED
110	01000000110011		<a href="#">Barriers</a>
110	01000001xx001x		UNALLOCATED
110	0100000xxx000x		UNALLOCATED
110	0100000xxx0100		<a href="#">PSTATE</a>
110	0100000xxx0101		UNALLOCATED
110	0100000xxx011x		UNALLOCATED
110	0100000xxx1xxx		UNALLOCATED
110	0100x01xxxxxxxxx		<a href="#">System instructions</a>
110	0100x1xxxxxxxxxx		<a href="#">System register move</a>
110	0101xxxxxxxxxxxx		UNALLOCATED
110	011xxxxxxxxxxxxx		UNALLOCATED
110	1xxxxxxxxxxxxxxxxx		<a href="#">Unconditional branch (register)</a>
x00			<a href="#">Unconditional branch (immediate)</a>
x01	0xxxxxxxxxxxxxxxxx		<a href="#">Compare and branch (immediate)</a>
x01	1xxxxxxxxxxxxxxxxx		<a href="#">Test and branch (immediate)</a>
x11			UNALLOCATED

### Conditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	o1	imm19																	o0	cond					

Decode fields		Instruction Details
o1	o0	
0	0	<a href="#">B.cond</a>
0	1	UNALLOCATED
1		UNALLOCATED

### Exception generation

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	opc		imm16																		op2		LL	

Decode fields			Instruction Details
opc	op2	LL	
	xx1		UNALLOCATED
	x1x		UNALLOCATED
	1xx		UNALLOCATED
000	000	01	SVC
000	000	10	<a href="#">HVC</a>
000	000	11	SMC
001	000	x1	UNALLOCATED
001	000	00	<a href="#">BRK</a>
001	000	1x	UNALLOCATED

Decode fields			Instruction Details
opc	op2	LL	
010	000	×1	UNALLOCATED
010	000	00	<a href="#">HLT</a>
010	000	1×	UNALLOCATED
011	000	01	UNALLOCATED
011	000	1×	UNALLOCATED
101	000	00	UNALLOCATED
101	000	01	<a href="#">DCPS1</a>
101	000	10	<a href="#">DCPS2</a>
101	000	11	<a href="#">DCPS3</a>
110	000		UNALLOCATED
111	000	01	UNALLOCATED
111	000	1×	UNALLOCATED

## Hints

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	CRm			op2			1	1	1	1	1	

Decode fields		Instruction Details	Architecture Version
CRm	op2		
		<a href="#">HINT</a>	-
0000	000	<a href="#">NOP</a>	-
0000	001	<a href="#">YIELD</a>	-
0000	010	<a href="#">WFE</a>	-
0000	011	<a href="#">WFI</a>	-
0000	100	<a href="#">SEV</a>	-
0000	101	<a href="#">SEVL</a>	-
0000	111	<a href="#">XPACD, XPACI, XPACLRI</a>	ARMv8.3
0001	000	<a href="#">PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA</a> — <a href="#">PACIA1716</a>	ARMv8.3
0001	010	<a href="#">PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB</a> — <a href="#">PACIB1716</a>	ARMv8.3
0001	100	<a href="#">AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA</a> — <a href="#">AUTIA1716</a>	ARMv8.3
0001	110	<a href="#">AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB</a> — <a href="#">AUTIB1716</a>	ARMv8.3
0010	000	<a href="#">ESB</a>	ARMv8.2
0010	001	<a href="#">PSB CSYNC</a>	ARMv8.2
0010	010	<a href="#">TSB CSYNC</a>	ARMv8.4
0010	100	<a href="#">CSDB</a>	-
0011	000	<a href="#">PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA</a> — <a href="#">PACIAZ</a>	ARMv8.3
0011	001	<a href="#">PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA</a> — <a href="#">PACIASP</a>	ARMv8.3
0011	010	<a href="#">PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB</a> — <a href="#">PACIBZ</a>	ARMv8.3
0011	011	<a href="#">PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB</a> — <a href="#">PACIBSP</a>	ARMv8.3
0011	100	<a href="#">AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA</a> — <a href="#">AUTIAZ</a>	ARMv8.3
0011	101	<a href="#">AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA</a> — <a href="#">AUTIASP</a>	ARMv8.3
0011	110	<a href="#">AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB</a> — <a href="#">AUTIBZ</a>	ARMv8.3
0011	111	<a href="#">AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB</a> — <a href="#">AUTIBSP</a>	ARMv8.3
0100	××0	<a href="#">BTI</a>	<a href="#">ARMv8.5</a>

Barriers

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm			op2			Rt					

Decode fields		Rt	Instruction Details
CRm	op2		
	000		UNALLOCATED
	001		UNALLOCATED
	010	11111	CLREX
	101	11111	<a href="#">DMB</a>
	110	11111	<a href="#">ISB</a>
	111	<b>!= 11111</b>	UNALLOCATED
	<b>111100</b>	11111	<a href="#">SBDSE</a>
<b>!= 0x000000</b>	100	11111	<a href="#">DSBSSBE</a>
<b>0000</b>	<b>100</b>	<b>11111</b>	<a href="#">SSBB</a>
<b>0001</b>	<b>011</b>		UNALLOCATED
<b>0001001x</b>	011		UNALLOCATED
<b>001x01xx</b>	011		UNALLOCATED
<b>01xx</b>	<b>011</b>		UNALLOCATED
<b>0100</b>	<b>100</b>	<b>11111</b>	<a href="#">PSSBE</a>
<b>0100</b>	<b>100</b>	<b>11111</b>	<a href="#">PSSBB</a>
<b>1xxx</b>	<b>011</b>		UNALLOCATED
<b>1xxx</b>	<b>011</b>		UNALLOCATED

PSTATE

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	op1			0			1	0	0	CRm			op2			Rt			

Decode fields		Rt	Instruction Details	Architecture Version
op1	op2			
		<b>!= 11111</b>	UNALLOCATED	-
		11111	<a href="#">MSR (immediate)</a>	-
000	000	11111	<a href="#">CFINV</a>	ARMv8.4
<b>000</b>	<b>001</b>	<b>11111</b>	<a href="#">XAFlag</a>	<b>ARMv8.5</b>
<b>000</b>	<b>010</b>	<b>11111</b>	<a href="#">AXFlag</a>	<b>ARMv8.5</b>

System instructions

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	L	0	1	op1			CRn			CRm			op2			Rt						

Decode fields		Instruction Details
L		
0		<a href="#">SYS</a>
1		SYSL

## System register move

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	L	1	o0	op1			CRn			CRm			op2			Rt						

Decode fields	Instruction Details
L	
0	MSR (register)
1	MRS

## Unconditional branch (register)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	opc			op2			op3			Rn			op4												

opc	op2	Decode fields op3	Rn	op4	Instruction Details	Architecture Version
	<b>!=</b> 11111				UNALLOCATED	!
				<b>!= 00000</b>	UNALLOCATED	!
0000	11111	000000		<b>!= 00000</b>	UNALLOCATED	!
		<b>!= 000000</b>			UNALLOCATED	!
0000	11111	000000		00000	BR	!
	<b>!=</b> 11111				UNALLOCATED	!
0000	11111	000001			UNALLOCATED	!
0000	11111	000000		00000	BR	!
0000	11111	000010		<b>!= 11111</b>	BRAA, BRAAZ, BRAB, BRABZ — key A, zero modifier	ARMv8.3
0000	11111	000010000011		11111	BRAA, BRAAZ, BRAB, BRABZ — key A, zero modifierkey B, zero modifier	ARMv8.3
00000001	11111	000011000000		<b>!=</b> 1111100000	BLR UNALLOCATED	-
00000001	11111	000011000010		11111	BRAA, BRAAZ, BRAB, BRABZ — key B, zero modifierkey A, zero modifier	ARMv8.3
00000001	11111	0001xx000011			BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, zero modifier	ARMv8.3
0000	11111	001xxx			UNALLOCATED	!
0010	11111	000000		00000	RET	!
00000010	11111	01xxxx000010			RETAA, RETAB — RETAA	ARMv8.3
00000010	11111	1xxxxx000011			RETAA, RETAB — RETAB	ARMv8.3
0001	11111	000000		<b>!= 00000</b>	UNALLOCATED	!
0011					UNALLOCATED	!
0001	11111	000000		00000	BLR	!
010x			<b>!=</b> 11111		UNALLOCATED	!



opc	op2	Decode fields op3	Rn	op4	Instruction Details	Architecture Version
0001	11111	000001			UNALLOCATED	!
0100	11111	000000	11111	00000	ERET	!
00010100	11111	000010		!= 11111	RETAA, RETAB — RETAA	ARMv8.3
00010100	11111	000010000011		11111	BLRAA, BLRAAZ, BLRAB, BLRABZ, RETAA, RETAB — key A, zero modifier, RETAB	ARMv8.3
0001	11111	000011		!= 11111	UNALLOCATED	!
0101	11111	000000	11111	00000	DRPS	!
0001	11111	000011		11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, zero modifier	ARMv8.3
011x					UNALLOCATED	!
00011xxx	11111	0001xx			UNALLOCATED	-
00011000	11111	001xxx000010			BRAA, BRAAZ, BRAB, BRABZ, UNALLOCATED — key A, register modifier	ARMv8.3
00011000	11111	01xxxx000011			BRAA, BRAAZ, BRAB, BRABZ, UNALLOCATED — key B, register modifier	ARMv8.3
00011001	11111	1xxxxx000010			BLRAA, BLRAAZ, BLRAB, BLRABZ, UNALLOCATED — key A, register modifier	ARMv8.3
00101001	11111	000000000011		!= 00000	BLRAA, BLRAAZ, BLRAB, BLRABZ, UNALLOCATED — key B, register modifier	ARMv8.3
0010	11111	000000		00000	RET	!
0010	11111	000001			UNALLOCATED	!
0010	11111	000010	!= 11111	!= 11111	UNALLOCATED	!
0010	11111	000010	11111	11111	RETAA, RETAB — RETAA	ARMv8.3
0010	11111	000011	!= 11111	!= 11111	UNALLOCATED	!
0010	11111	000011	11111	11111	RETAA, RETAB — RETAB	ARMv8.3
0010	11111	0001xx			UNALLOCATED	!
0010	11111	001xxx			UNALLOCATED	!
0010	11111	01xxxx			UNALLOCATED	!
0010	11111	1xxxxx			UNALLOCATED	!
0011	11111				UNALLOCATED	!
0100	11111	000000	!= 11111	!= 00000	UNALLOCATED	!
0100	11111	000000	!= 11111	00000	UNALLOCATED	!
0100	11111	000000	11111	!= 00000	UNALLOCATED	!
0100	11111	000000	11111	00000	ERET	!
0100	11111	000001			UNALLOCATED	!
0100	11111	000010	!= 11111	!= 11111	UNALLOCATED	!
0100	11111	000010	!= 11111	11111	UNALLOCATED	!
0100	11111	000010	11111	!= 11111	UNALLOCATED	!
0100	11111	000010	11111	11111	RETAA, RETAB — RETAA	ARMv8.3

		Decode fields				Instruction Details		Architecture Version
opc	op2	op3	Rn	op4				
0100	11111	000011	!= 11111	!= 11111	UNALLOCATED			
0100	11111	000011	!= 11111	11111	UNALLOCATED			
0100	11111	000011	11111	!= 11111	UNALLOCATED			
0100	11111	000011	11111	11111	ERETAA, ERETAB — ERETAB			ARMv8.3
0100	11111	0001xx			UNALLOCATED			
0100	11111	001xxx			UNALLOCATED			
0100	11111	01xxxx			UNALLOCATED			
0100	11111	1xxxxx			UNALLOCATED			
0101	11111	!= 000000			UNALLOCATED			
0101	11111	000000	!= 11111	!= 00000	UNALLOCATED			
0101	11111	000000	!= 11111	00000	UNALLOCATED			
0101	11111	000000	11111	!= 00000	UNALLOCATED			
0101	11111	000000	11111	00000	DRPS			
011x	11111				UNALLOCATED			
1000	11111	00000x			UNALLOCATED			
1000	11111	000010			BRAA, BRAAZ, BRAB, BRABZ — key A, register modifier			ARMv8.3
1000	11111	000011			BRAA, BRAAZ, BRAB, BRABZ — key B, register modifier			ARMv8.3
1000	11111	0001xx			UNALLOCATED			
1000	11111	001xxx			UNALLOCATED			
1000	11111	01xxxx			UNALLOCATED			
1000	11111	1xxxxx			UNALLOCATED			
1001	11111	00000x			UNALLOCATED			
1001	11111	000010			BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, register modifier			ARMv8.3
1001	11111	000011			BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, register modifier			ARMv8.3
1001	11111	0001xx			UNALLOCATED			
1001	11111	001xxx			UNALLOCATED			
1001	11111	01xxxx			UNALLOCATED			
1001	11111	1xxxxx			UNALLOCATED			
101x	11111				UNALLOCATED			
11xx	11111				UNALLOCATED			

Unconditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
op	0	0	1	0	1	imm26																										

Decode fields		Instruction Details	
op			
0		B	
1		BL	

## Compare and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	0	1	0	op	imm19																				Rt			

Decode fields		Instruction Details
sf	op	
0	0	<a href="#">CBZ</a> — <a href="#">32-bit</a>
0	1	<a href="#">CBNZ</a> — <a href="#">32-bit</a>
1	0	<a href="#">CBZ</a> — <a href="#">64-bit</a>
1	1	<a href="#">CBNZ</a> — <a href="#">64-bit</a>

## Test and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
b5	0	1	1	0	1	1	op	b40					imm14										Rt								

Decode fields		Instruction Details
op		
0		<a href="#">TBZ</a>
1		<a href="#">TBNZ</a>

## Loads and Stores

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0				1	op1		0	op2		op3						op4															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0				op1		1	op2		0	op3		op4						op5													

Decode fields					Instruction details
op0	op1	op2	op3	op4	
0x00	1	00	000000		<a href="#">Advanced SIMD load/store multiple structures</a>
0x00	1	01	0xxxxx		<a href="#">Advanced SIMD load/store multiple structures (post-indexed)</a>
0x00	1	0x	1xxxxx		UNALLOCATED
0x00	1	10	x00000		<a href="#">Advanced SIMD load/store single structure</a>
0x00	1	11			<a href="#">Advanced SIMD load/store single structure (post-indexed)</a>
0x00	1	x0	x1xxxx		UNALLOCATED
0x00	1	x0	xx1xxx		UNALLOCATED
0x00	1	x0	xxx1xx		UNALLOCATED
0x00	1	x0	xxxx1x		UNALLOCATED
0x00	1	x0	xxxxx1		UNALLOCATED
0x01	0	1x	1xxxxx		UNALLOCATED
1001	0	1x	1xxxxx		UNALLOCATED
1101	0	1x	1xxxxx		<a href="#">Load/store memory tags</a>
1x00	1				UNALLOCATED
xx00	0	0x			<a href="#">Load/store exclusive</a>
xx00	0	1x			UNALLOCATED
xx01		0x			<a href="#">Load register (literal)</a>
xx01		1x	0xxxxx	00	UNALLOCATED

xx10		00			<a href="#">Load/store no-allocate pair (offset)</a>
xx10		01			<a href="#">Load/store register pair (post-indexed)</a>
xx10		10			<a href="#">Load/store register pair (offset)</a>
xx10		11			<a href="#">Load/store register pair (pre-indexed)</a>
xx11		0x	0xxxxx	00	<a href="#">Load/store register (unscaled immediate)</a>
xx11		0x	0xxxxx	01	<a href="#">Load/store register (immediate post-indexed)</a>
xx11		0x	0xxxxx	10	<a href="#">Load/store register (unprivileged)</a>
xx11		0x	0xxxxx	11	<a href="#">Load/store register (immediate pre-indexed)</a>
xx11		0x	1xxxxx	00	<a href="#">Atomic memory operations</a>
xx11		0x	1xxxxx	10	<a href="#">Load/store register (register offset)</a>
xx11		0x	1xxxxx	x1	<a href="#">Load/store register (pac)</a>
xx11		1x			<a href="#">Load/store register (unsigned immediate)</a>

Decode fields					Instruction details	
op0	op1	op2	op3	op4	op5	
0	00	1	00	000000		<a href="#">Advanced SIMD load/store multiple structures</a>
0	00	1	01	0xxxxx		<a href="#">Advanced SIMD load/store multiple structures (post-indexed)</a>
0	00	1	0x	1xxxxx		UNALLOCATED
0	00	1	10	x00000		<a href="#">Advanced SIMD load/store single structure</a>
0	00	1	11			<a href="#">Advanced SIMD load/store single structure (post-indexed)</a>
0	00	1	x0	x1xxxx		UNALLOCATED
0	00	1	x0	xx1xxx		UNALLOCATED
0	00	1	x0	xxx1xx		UNALLOCATED
0	00	1	x0	xxxx1x		UNALLOCATED
0	00	1	x0	xxxxx1		UNALLOCATED
1	00	1				UNALLOCATED
	00	0	0x			<a href="#">Load/store exclusive</a>
	00	0	1x			UNALLOCATED
	01	0	1x	0xxxxx	00	<a href="#">LDAPR/STLR (unscaled immediate)</a>
	01	1	1x	0xxxxx	00	UNALLOCATED
	01		0x			<a href="#">Load register (literal)</a>
	01		1x	0xxxxx	01	UNALLOCATED
	01		1x	0xxxxx	10	UNALLOCATED
	01		1x	0xxxxx	11	UNALLOCATED
	01		1x	1xxxxx		UNALLOCATED
	10		00			<a href="#">Load/store no-allocate pair (offset)</a>
	10		01			<a href="#">Load/store register pair (post-indexed)</a>
	10		10			<a href="#">Load/store register pair (offset)</a>
	10		11			<a href="#">Load/store register pair (pre-indexed)</a>
	11		0x	0xxxxx	00	<a href="#">Load/store register (unscaled immediate)</a>
	11		0x	0xxxxx	01	<a href="#">Load/store register (immediate post-indexed)</a>
	11		0x	0xxxxx	10	<a href="#">Load/store register (unprivileged)</a>
	11		0x	0xxxxx	11	<a href="#">Load/store register (immediate pre-indexed)</a>
	11		0x	1xxxxx	00	<a href="#">Atomic memory operations</a>
	11		0x	1xxxxx	10	<a href="#">Load/store register (register offset)</a>
	11		0x	1xxxxx	x1	<a href="#">Load/store register (pac)</a>
	11		1x			<a href="#">Load/store register (unsigned immediate)</a>

## Advanced SIMD load/store fields multiple structures

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	L	0	0	0	0	0	0	opcode					size		Rn				Rt				

Decode fields		Instruction Details	
L	opcode		
0	0000	<a href="#">ST4 (multiple structures)</a>	
0	0001	UNALLOCATED	
0	0010	<a href="#">ST1 (multiple structures)</a> — <a href="#">four registers</a>	
0	0011	UNALLOCATED	
0	0100	<a href="#">ST3 (multiple structures)</a>	
0	0101	UNALLOCATED	
0	0110	<a href="#">ST1 (multiple structures)</a> — <a href="#">three registers</a>	
0	0111	<a href="#">ST1 (multiple structures)</a> — <a href="#">one register</a>	
0	1000	<a href="#">ST2 (multiple structures)</a>	
0	1001	UNALLOCATED	
0	1010	<a href="#">ST1 (multiple structures)</a> — <a href="#">two registers</a>	
0	1011	UNALLOCATED	
0	11xx	UNALLOCATED	
1	0000	<a href="#">LD4 (multiple structures)</a>	
1	0001	UNALLOCATED	
1	0010	<a href="#">LD1 (multiple structures)</a> — <a href="#">four registers</a>	
1	0011	UNALLOCATED	
1	0100	<a href="#">LD3 (multiple structures)</a>	
1	0101	UNALLOCATED	
1	0110	<a href="#">LD1 (multiple structures)</a> — <a href="#">three registers</a>	
1	0111	<a href="#">LD1 (multiple structures)</a> — <a href="#">one register</a>	
1	1000	<a href="#">LD2 (multiple structures)</a>	
1	1001	UNALLOCATED	
1	1010	<a href="#">LD1 (multiple structures)</a> — <a href="#">two registers</a>	
1	1011	UNALLOCATED	
1	11xx	UNALLOCATED	

### Advanced SIMD load/store multiple structures (post-indexed)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	L	0	Rm					opcode					size		Rn				Rt				

Decode fields		Instruction Details	
L	Rm	opcode	
0		0001	UNALLOCATED
0		0011	UNALLOCATED
0		0101	UNALLOCATED
0		1001	UNALLOCATED
0		1011	UNALLOCATED
0		11xx	UNALLOCATED
0	!= 11111	0000	<a href="#">ST4 (multiple structures)</a> — <a href="#">register offset</a>
0	!= 11111	0010	<a href="#">ST1 (multiple structures)</a> — <a href="#">four registers, register offset</a>
0	!= 11111	0100	<a href="#">ST3 (multiple structures)</a> — <a href="#">register offset</a>
0	!= 11111	0110	<a href="#">ST1 (multiple structures)</a> — <a href="#">three registers, register offset</a>
0	!= 11111	0111	<a href="#">ST1 (multiple structures)</a> — <a href="#">one register, register offset</a>

Decode fields			Instruction Details
L	Rm	opcode	
0	!= 11111	1000	<a href="#">ST2 (multiple structures)</a> — <a href="#">register offset</a>
0	!= 11111	1010	<a href="#">ST1 (multiple structures)</a> — <a href="#">two registers, register offset</a>
0	11111	0000	<a href="#">ST4 (multiple structures)</a> — <a href="#">immediate offset</a>
0	11111	0010	<a href="#">ST1 (multiple structures)</a> — <a href="#">four registers, immediate offset</a>
0	11111	0100	<a href="#">ST3 (multiple structures)</a> — <a href="#">immediate offset</a>
0	11111	0110	<a href="#">ST1 (multiple structures)</a> — <a href="#">three registers, immediate offset</a>
0	11111	0111	<a href="#">ST1 (multiple structures)</a> — <a href="#">one register, immediate offset</a>
0	11111	1000	<a href="#">ST2 (multiple structures)</a> — <a href="#">immediate offset</a>
0	11111	1010	<a href="#">ST1 (multiple structures)</a> — <a href="#">two registers, immediate offset</a>
1		0001	UNALLOCATED
1		0011	UNALLOCATED
1		0101	UNALLOCATED
1		1001	UNALLOCATED
1		1011	UNALLOCATED
1		11xx	UNALLOCATED
1	!= 11111	0000	<a href="#">LD4 (multiple structures)</a> — <a href="#">register offset</a>
1	!= 11111	0010	<a href="#">LD1 (multiple structures)</a> — <a href="#">four registers, register offset</a>
1	!= 11111	0100	<a href="#">LD3 (multiple structures)</a> — <a href="#">register offset</a>
1	!= 11111	0110	<a href="#">LD1 (multiple structures)</a> — <a href="#">three registers, register offset</a>
1	!= 11111	0111	<a href="#">LD1 (multiple structures)</a> — <a href="#">one register, register offset</a>
1	!= 11111	1000	<a href="#">LD2 (multiple structures)</a> — <a href="#">register offset</a>
1	!= 11111	1010	<a href="#">LD1 (multiple structures)</a> — <a href="#">two registers, register offset</a>
1	11111	0000	<a href="#">LD4 (multiple structures)</a> — <a href="#">immediate offset</a>
1	11111	0010	<a href="#">LD1 (multiple structures)</a> — <a href="#">four registers, immediate offset</a>
1	11111	0100	<a href="#">LD3 (multiple structures)</a> — <a href="#">immediate offset</a>
1	11111	0110	<a href="#">LD1 (multiple structures)</a> — <a href="#">three registers, immediate offset</a>
1	11111	0111	<a href="#">LD1 (multiple structures)</a> — <a href="#">one register, immediate offset</a>
1	11111	1000	<a href="#">LD2 (multiple structures)</a> — <a href="#">immediate offset</a>
1	11111	1010	<a href="#">LD1 (multiple structures)</a> — <a href="#">two registers, immediate offset</a>

## Advanced SIMD load/store single structure

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	L	R	0	0	0	0	0	opcode	S	size						Rn						Rt	

Decode fields				Instruction Details	
L	R	opcode	S	size	
0		11x			UNALLOCATED
0	0	000			<a href="#">ST1 (single structure)</a> — <a href="#">8-bit</a>
0	0	001			<a href="#">ST3 (single structure)</a> — <a href="#">8-bit</a>
0	0	010		x0	<a href="#">ST1 (single structure)</a> — <a href="#">16-bit</a>
0	0	010		x1	UNALLOCATED
0	0	011		x0	<a href="#">ST3 (single structure)</a> — <a href="#">16-bit</a>
0	0	011		x1	UNALLOCATED
0	0	100		00	<a href="#">ST1 (single structure)</a> — <a href="#">32-bit</a>
0	0	100		1x	UNALLOCATED
0	0	100	0	01	<a href="#">ST1 (single structure)</a> — <a href="#">64-bit</a>

Decode fields					Instruction Details
L	R	opcode	S	size	
0	0	100	1	01	UNALLOCATED
0	0	101		00	<a href="#">ST3 (single structure)</a> — <a href="#">32-bit</a>
0	0	101		10	UNALLOCATED
0	0	101	0	01	<a href="#">ST3 (single structure)</a> — <a href="#">64-bit</a>
0	0	101	0	11	UNALLOCATED
0	0	101	1	×1	UNALLOCATED
0	1	000			<a href="#">ST2 (single structure)</a> — <a href="#">8-bit</a>
0	1	001			<a href="#">ST4 (single structure)</a> — <a href="#">8-bit</a>
0	1	010		×0	<a href="#">ST2 (single structure)</a> — <a href="#">16-bit</a>
0	1	010		×1	UNALLOCATED
0	1	011		×0	<a href="#">ST4 (single structure)</a> — <a href="#">16-bit</a>
0	1	011		×1	UNALLOCATED
0	1	100		00	<a href="#">ST2 (single structure)</a> — <a href="#">32-bit</a>
0	1	100		10	UNALLOCATED
0	1	100	0	01	<a href="#">ST2 (single structure)</a> — <a href="#">64-bit</a>
0	1	100	0	11	UNALLOCATED
0	1	100	1	×1	UNALLOCATED
0	1	101		00	<a href="#">ST4 (single structure)</a> — <a href="#">32-bit</a>
0	1	101		10	UNALLOCATED
0	1	101	0	01	<a href="#">ST4 (single structure)</a> — <a href="#">64-bit</a>
0	1	101	0	11	UNALLOCATED
0	1	101	1	×1	UNALLOCATED
1	0	000			<a href="#">LD1 (single structure)</a> — <a href="#">8-bit</a>
1	0	001			<a href="#">LD3 (single structure)</a> — <a href="#">8-bit</a>
1	0	010		×0	<a href="#">LD1 (single structure)</a> — <a href="#">16-bit</a>
1	0	010		×1	UNALLOCATED
1	0	011		×0	<a href="#">LD3 (single structure)</a> — <a href="#">16-bit</a>
1	0	011		×1	UNALLOCATED
1	0	100		00	<a href="#">LD1 (single structure)</a> — <a href="#">32-bit</a>
1	0	100		1×	UNALLOCATED
1	0	100	0	01	<a href="#">LD1 (single structure)</a> — <a href="#">64-bit</a>
1	0	100	1	01	UNALLOCATED
1	0	101		00	<a href="#">LD3 (single structure)</a> — <a href="#">32-bit</a>
1	0	101		10	UNALLOCATED
1	0	101	0	01	<a href="#">LD3 (single structure)</a> — <a href="#">64-bit</a>
1	0	101	0	11	UNALLOCATED
1	0	101	1	×1	UNALLOCATED
1	0	110	0		<a href="#">LD1R</a>
1	0	110	1		UNALLOCATED
1	0	111	0		<a href="#">LD3R</a>
1	0	111	1		UNALLOCATED
1	1	000			<a href="#">LD2 (single structure)</a> — <a href="#">8-bit</a>
1	1	001			<a href="#">LD4 (single structure)</a> — <a href="#">8-bit</a>
1	1	010		×0	<a href="#">LD2 (single structure)</a> — <a href="#">16-bit</a>
1	1	010		×1	UNALLOCATED
1	1	011		×0	<a href="#">LD4 (single structure)</a> — <a href="#">16-bit</a>
1	1	011		×1	UNALLOCATED

Decode fields					Instruction Details
L	R	opcode	S	size	
1	1	100		00	<a href="#">LD2 (single structure)</a> — <a href="#">32-bit</a>
1	1	100		10	UNALLOCATED
1	1	100	0	01	<a href="#">LD2 (single structure)</a> — <a href="#">64-bit</a>
1	1	100	0	11	UNALLOCATED
1	1	100	1	×1	UNALLOCATED
1	1	101		00	<a href="#">LD4 (single structure)</a> — <a href="#">32-bit</a>
1	1	101		10	UNALLOCATED
1	1	101	0	01	<a href="#">LD4 (single structure)</a> — <a href="#">64-bit</a>
1	1	101	0	11	UNALLOCATED
1	1	101	1	×1	UNALLOCATED
1	1	110	0		<a href="#">LD2R</a>
1	1	110	1		UNALLOCATED
1	1	111	0		<a href="#">LD4R</a>
1	1	111	1		UNALLOCATED

### Advanced SIMD load/store single structure (post-indexed)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	L	R	Rm				opcode		S	size		Rn				Rt							

Decode fields					Instruction Details	
L	R	Rm	opcode	S	size	
0			11x			UNALLOCATED
0	0		010		x1	UNALLOCATED
0	0		011		x1	UNALLOCATED
0	0		100		1x	UNALLOCATED
0	0		100	1	01	UNALLOCATED
0	0		101		10	UNALLOCATED
0	0		101	0	11	UNALLOCATED
0	0		101	1	x1	UNALLOCATED
0	0	!= 11111	000			<a href="#">ST1 (single structure)</a> — <a href="#">8-bit, register offset</a>
0	0	!= 11111	001			<a href="#">ST3 (single structure)</a> — <a href="#">8-bit, register offset</a>
0	0	!= 11111	010		x0	<a href="#">ST1 (single structure)</a> — <a href="#">16-bit, register offset</a>
0	0	!= 11111	011		x0	<a href="#">ST3 (single structure)</a> — <a href="#">16-bit, register offset</a>
0	0	!= 11111	100		00	<a href="#">ST1 (single structure)</a> — <a href="#">32-bit, register offset</a>
0	0	!= 11111	100	0	01	<a href="#">ST1 (single structure)</a> — <a href="#">64-bit, register offset</a>
0	0	!= 11111	101		00	<a href="#">ST3 (single structure)</a> — <a href="#">32-bit, register offset</a>
0	0	!= 11111	101	0	01	<a href="#">ST3 (single structure)</a> — <a href="#">64-bit, register offset</a>
0	0	11111	000			<a href="#">ST1 (single structure)</a> — <a href="#">8-bit, immediate offset</a>
0	0	11111	001			<a href="#">ST3 (single structure)</a> — <a href="#">8-bit, immediate offset</a>
0	0	11111	010		x0	<a href="#">ST1 (single structure)</a> — <a href="#">16-bit, immediate offset</a>
0	0	11111	011		x0	<a href="#">ST3 (single structure)</a> — <a href="#">16-bit, immediate offset</a>
0	0	11111	100		00	<a href="#">ST1 (single structure)</a> — <a href="#">32-bit, immediate offset</a>
0	0	11111	100	0	01	<a href="#">ST1 (single structure)</a> — <a href="#">64-bit, immediate offset</a>
0	0	11111	101		00	<a href="#">ST3 (single structure)</a> — <a href="#">32-bit, immediate offset</a>
0	0	11111	101	0	01	<a href="#">ST3 (single structure)</a> — <a href="#">64-bit, immediate offset</a>
0	1		010		x1	UNALLOCATED



Decode fields						Instruction Details
L	R	Rm	opcode	S	size	
0	1		011		×1	UNALLOCATED
0	1		100		10	UNALLOCATED
0	1		100	0	11	UNALLOCATED
0	1		100	1	×1	UNALLOCATED
0	1		101		10	UNALLOCATED
0	1		101	0	11	UNALLOCATED
0	1		101	1	×1	UNALLOCATED
0	1	!= 11111	000			<a href="#">ST2 (single structure)</a> — <a href="#">8-bit, register offset</a>
0	1	!= 11111	001			<a href="#">ST4 (single structure)</a> — <a href="#">8-bit, register offset</a>
0	1	!= 11111	010		×0	<a href="#">ST2 (single structure)</a> — <a href="#">16-bit, register offset</a>
0	1	!= 11111	011		×0	<a href="#">ST4 (single structure)</a> — <a href="#">16-bit, register offset</a>
0	1	!= 11111	100		00	<a href="#">ST2 (single structure)</a> — <a href="#">32-bit, register offset</a>
0	1	!= 11111	100	0	01	<a href="#">ST2 (single structure)</a> — <a href="#">64-bit, register offset</a>
0	1	!= 11111	101		00	<a href="#">ST4 (single structure)</a> — <a href="#">32-bit, register offset</a>
0	1	!= 11111	101	0	01	<a href="#">ST4 (single structure)</a> — <a href="#">64-bit, register offset</a>
0	1	11111	000			<a href="#">ST2 (single structure)</a> — <a href="#">8-bit, immediate offset</a>
0	1	11111	001			<a href="#">ST4 (single structure)</a> — <a href="#">8-bit, immediate offset</a>
0	1	11111	010		×0	<a href="#">ST2 (single structure)</a> — <a href="#">16-bit, immediate offset</a>
0	1	11111	011		×0	<a href="#">ST4 (single structure)</a> — <a href="#">16-bit, immediate offset</a>
0	1	11111	100		00	<a href="#">ST2 (single structure)</a> — <a href="#">32-bit, immediate offset</a>
0	1	11111	100	0	01	<a href="#">ST2 (single structure)</a> — <a href="#">64-bit, immediate offset</a>
0	1	11111	101		00	<a href="#">ST4 (single structure)</a> — <a href="#">32-bit, immediate offset</a>
0	1	11111	101	0	01	<a href="#">ST4 (single structure)</a> — <a href="#">64-bit, immediate offset</a>
1	0		010		×1	UNALLOCATED
1	0		011		×1	UNALLOCATED
1	0		100		1×	UNALLOCATED
1	0		100	1	01	UNALLOCATED
1	0		101		10	UNALLOCATED
1	0		101	0	11	UNALLOCATED
1	0		101	1	×1	UNALLOCATED
1	0		110	1		UNALLOCATED
1	0		111	1		UNALLOCATED
1	0	!= 11111	000			<a href="#">LD1 (single structure)</a> — <a href="#">8-bit, register offset</a>
1	0	!= 11111	001			<a href="#">LD3 (single structure)</a> — <a href="#">8-bit, register offset</a>
1	0	!= 11111	010		×0	<a href="#">LD1 (single structure)</a> — <a href="#">16-bit, register offset</a>
1	0	!= 11111	011		×0	<a href="#">LD3 (single structure)</a> — <a href="#">16-bit, register offset</a>
1	0	!= 11111	100		00	<a href="#">LD1 (single structure)</a> — <a href="#">32-bit, register offset</a>
1	0	!= 11111	100	0	01	<a href="#">LD1 (single structure)</a> — <a href="#">64-bit, register offset</a>
1	0	!= 11111	101		00	<a href="#">LD3 (single structure)</a> — <a href="#">32-bit, register offset</a>
1	0	!= 11111	101	0	01	<a href="#">LD3 (single structure)</a> — <a href="#">64-bit, register offset</a>
1	0	!= 11111	110	0		<a href="#">LD1R</a> — <a href="#">register offset</a>
1	0	!= 11111	111	0		<a href="#">LD3R</a> — <a href="#">register offset</a>
1	0	11111	000			<a href="#">LD1 (single structure)</a> — <a href="#">8-bit, immediate offset</a>
1	0	11111	001			<a href="#">LD3 (single structure)</a> — <a href="#">8-bit, immediate offset</a>
1	0	11111	010		×0	<a href="#">LD1 (single structure)</a> — <a href="#">16-bit, immediate offset</a>
1	0	11111	011		×0	<a href="#">LD3 (single structure)</a> — <a href="#">16-bit, immediate offset</a>
1	0	11111	100		00	<a href="#">LD1 (single structure)</a> — <a href="#">32-bit, immediate offset</a>

Decode fields						Instruction Details
L	R	Rm	opcode	S	size	
1	0	11111	100	0	01	<a href="#">LD1 (single structure)</a> — <a href="#">64-bit, immediate offset</a>
1	0	11111	101		00	<a href="#">LD3 (single structure)</a> — <a href="#">32-bit, immediate offset</a>
1	0	11111	101	0	01	<a href="#">LD3 (single structure)</a> — <a href="#">64-bit, immediate offset</a>
1	0	11111	110	0		<a href="#">LD1R</a> — <a href="#">immediate offset</a>
1	0	11111	111	0		<a href="#">LD3R</a> — <a href="#">immediate offset</a>
1	1		010		×1	UNALLOCATED
1	1		011		×1	UNALLOCATED
1	1		100		10	UNALLOCATED
1	1		100	0	11	UNALLOCATED
1	1		100	1	×1	UNALLOCATED
1	1		101		10	UNALLOCATED
1	1		101	0	11	UNALLOCATED
1	1		101	1	×1	UNALLOCATED
1	1		110	1		UNALLOCATED
1	1		111	1		UNALLOCATED
1	1	!= 11111	000			<a href="#">LD2 (single structure)</a> — <a href="#">8-bit, register offset</a>
1	1	!= 11111	001			<a href="#">LD4 (single structure)</a> — <a href="#">8-bit, register offset</a>
1	1	!= 11111	010		×0	<a href="#">LD2 (single structure)</a> — <a href="#">16-bit, register offset</a>
1	1	!= 11111	011		×0	<a href="#">LD4 (single structure)</a> — <a href="#">16-bit, register offset</a>
1	1	!= 11111	100		00	<a href="#">LD2 (single structure)</a> — <a href="#">32-bit, register offset</a>
1	1	!= 11111	100	0	01	<a href="#">LD2 (single structure)</a> — <a href="#">64-bit, register offset</a>
1	1	!= 11111	101		00	<a href="#">LD4 (single structure)</a> — <a href="#">32-bit, register offset</a>
1	1	!= 11111	101	0	01	<a href="#">LD4 (single structure)</a> — <a href="#">64-bit, register offset</a>
1	1	!= 11111	110	0		<a href="#">LD2R</a> — <a href="#">register offset</a>
1	1	!= 11111	111	0		<a href="#">LD4R</a> — <a href="#">register offset</a>
1	1	11111	000			<a href="#">LD2 (single structure)</a> — <a href="#">8-bit, immediate offset</a>
1	1	11111	001			<a href="#">LD4 (single structure)</a> — <a href="#">8-bit, immediate offset</a>
1	1	11111	010		×0	<a href="#">LD2 (single structure)</a> — <a href="#">16-bit, immediate offset</a>
1	1	11111	011		×0	<a href="#">LD4 (single structure)</a> — <a href="#">16-bit, immediate offset</a>
1	1	11111	100		00	<a href="#">LD2 (single structure)</a> — <a href="#">32-bit, immediate offset</a>
1	1	11111	100	0	01	<a href="#">LD2 (single structure)</a> — <a href="#">64-bit, immediate offset</a>
1	1	11111	101		00	<a href="#">LD4 (single structure)</a> — <a href="#">32-bit, immediate offset</a>
1	1	11111	101	0	01	<a href="#">LD4 (single structure)</a> — <a href="#">64-bit, immediate offset</a>
1	1	11111	110	0		<a href="#">LD2R</a> — <a href="#">immediate offset</a>
1	1	11111	111	0		<a href="#">LD4R</a> — <a href="#">immediate offset</a>

**Load/store memory tagexclusive**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	opc	1											op2				Rn					Rt		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	0	1	0	0	0	0	e2	L	e4				Rs		e0				Rt2				Rn					Rt		

Decode fields		Instruction Details		Architecture Version
opc	imm9	op2		
00		00	UNALLOCATED	-
00		01	<a href="#">STG</a> — <a href="#">post-index</a>	ARMv8.5

Decode fields			Instruction Details	Architecture Version
opc	imm9	op2		
00		10	<a href="#">STG — signed offset</a>	ARMv8.5
00		11	<a href="#">STG — pre-index</a>	ARMv8.5
01		00	<a href="#">LDG</a>	ARMv8.5
01		01	<a href="#">STZG — post-index</a>	ARMv8.5
01		10	<a href="#">STZG — signed offset</a>	ARMv8.5
01		11	<a href="#">STZG — pre-index</a>	ARMv8.5
10		01	<a href="#">ST2G — post-index</a>	ARMv8.5
10		10	<a href="#">ST2G — signed offset</a>	ARMv8.5
10		11	<a href="#">ST2G — pre-index</a>	ARMv8.5
10	!= 000000000	00	UNALLOCATED	—
10	000000000	00	<a href="#">STGV</a>	ARMv8.5
11		01	<a href="#">STZ2G — post-index</a>	ARMv8.5
11		10	<a href="#">STZ2G — signed offset</a>	ARMv8.5
11		11	<a href="#">STZ2G — pre-index</a>	ARMv8.5
11	!= 000000000	00	UNALLOCATED	—
11	000000000	00	<a href="#">LDGV</a>	ARMv8.5

size	Decode fields					Rt2	Instruction Details	Architecture Version
	o2	L	o1	o0				
	1		1		!= 11111		UNALLOCATED	—
0x	0		1		!= 11111		UNALLOCATED	—
00	0	0	0	0			<a href="#">STXRB</a>	—
00	0	0	0	1			<a href="#">STLXRB</a>	—
00	0	0	1	0	11111		<a href="#">CASP, CASPA, CASPAL, CASPL — 32-bit CASP</a>	ARMv8.1
00	0	0	1	1	11111		<a href="#">CASP, CASPA, CASPAL, CASPL — 32-bit CASPL</a>	ARMv8.1
00	0	1	0	0			<a href="#">LDXRB</a>	—
00	0	1	0	1			<a href="#">LDAXRB</a>	—
00	0	1	1	0	11111		<a href="#">CASP, CASPA, CASPAL, CASPL — 32-bit CASPA</a>	ARMv8.1
00	0	1	1	1	11111		<a href="#">CASP, CASPA, CASPAL, CASPL — 32-bit CASPAL</a>	ARMv8.1
00	1	0	0	0			<a href="#">STLLRB</a>	ARMv8.1
00	1	0	0	1			<a href="#">STLRB</a>	—
00	1	0	1	0	11111		<a href="#">CASB, CASAB, CASALB, CASLB — CASB</a>	ARMv8.1
00	1	0	1	1	11111		<a href="#">CASB, CASAB, CASALB, CASLB — CASLB</a>	ARMv8.1
00	1	1	0	0			<a href="#">LDLARB</a>	ARMv8.1
00	1	1	0	1			<a href="#">LDARB</a>	—
00	1	1	1	0	11111		<a href="#">CASB, CASAB, CASALB, CASLB — CASAB</a>	ARMv8.1
00	1	1	1	1	11111		<a href="#">CASB, CASAB, CASALB, CASLB — CASALB</a>	ARMv8.1
01	0	0	0	0			<a href="#">STXRH</a>	—
01	0	0	0	1			<a href="#">STLXRH</a>	—
01	0	0	1	0	11111		<a href="#">CASP, CASPA, CASPAL, CASPL — 64-bit CASP</a>	ARMv8.1
01	0	0	1	1	11111		<a href="#">CASP, CASPA, CASPAL, CASPL — 64-bit CASPL</a>	ARMv8.1
01	0	1	0	0			<a href="#">LDXRH</a>	—
01	0	1	0	1			<a href="#">LDAXRH</a>	—
01	0	1	1	0	11111		<a href="#">CASP, CASPA, CASPAL, CASPL — 64-bit CASPA</a>	ARMv8.1
01	0	1	1	1	11111		<a href="#">CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL</a>	ARMv8.1
01	1	0	0	0			<a href="#">STLLRH</a>	ARMv8.1
01	1	0	0	1			<a href="#">STLRH</a>	—
01	1	0	1	0	11111		<a href="#">CASH, CASH, CASALH, CASLH — CASH</a>	ARMv8.1
01	1	0	1	1	11111		<a href="#">CASH, CASH, CASALH, CASLH — CASLH</a>	ARMv8.1

Decode fields					Rt2	Instruction Details	Architecture Version
size	o2	L	o1	o0			
01	1	1	0	0		<a href="#">LDLARH</a>	ARMv8.1
01	1	1	0	1		<a href="#">LDARH</a>	!
01	1	1	1	0	11111	<a href="#">CASH, CASH, CASALH, CASLH — CASH</a>	ARMv8.1
01	1	1	1	1	11111	<a href="#">CASH, CASH, CASALH, CASLH — CASALH</a>	ARMv8.1
10	0	0	0	0		<a href="#">STXR — 32-bit</a>	!
10	0	0	0	1		<a href="#">STLXR — 32-bit</a>	!
10	0	0	1	0		<a href="#">STXP — 32-bit</a>	!
10	0	0	1	1		<a href="#">STLXP — 32-bit</a>	!
10	0	1	0	0		<a href="#">LDXR — 32-bit</a>	!
10	0	1	0	1		<a href="#">LDAXR — 32-bit</a>	!
10	0	1	1	0		<a href="#">LDXP — 32-bit</a>	!
10	0	1	1	1		<a href="#">LDAXP — 32-bit</a>	!
10	1	0	0	0		<a href="#">STLLR — 32-bit</a>	ARMv8.1
10	1	0	0	1		<a href="#">STLR — 32-bit</a>	!
10	1	0	1	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 32-bit CAS</a>	ARMv8.1
10	1	0	1	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 32-bit CASL</a>	ARMv8.1
10	1	1	0	0		<a href="#">LDLAR — 32-bit</a>	ARMv8.1
10	1	1	0	1		<a href="#">LDAR — 32-bit</a>	!
10	1	1	1	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 32-bit CASA</a>	ARMv8.1
10	1	1	1	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 32-bit CASAL</a>	ARMv8.1
11	0	0	0	0		<a href="#">STXR — 64-bit</a>	!
11	0	0	0	1		<a href="#">STLXR — 64-bit</a>	!
11	0	0	1	0		<a href="#">STXP — 64-bit</a>	!
11	0	0	1	1		<a href="#">STLXP — 64-bit</a>	!
11	0	1	0	0		<a href="#">LDXR — 64-bit</a>	!
11	0	1	0	1		<a href="#">LDAXR — 64-bit</a>	!
11	0	1	1	0		<a href="#">LDXP — 64-bit</a>	!
11	0	1	1	1		<a href="#">LDAXP — 64-bit</a>	!
11	1	0	0	0		<a href="#">STLLR — 64-bit</a>	ARMv8.1
11	1	0	0	1		<a href="#">STLR — 64-bit</a>	!
11	1	0	1	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CAS</a>	ARMv8.1
11	1	0	1	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CASL</a>	ARMv8.1
11	1	1	0	0		<a href="#">LDLAR — 64-bit</a>	ARMv8.1
11	1	1	0	1		<a href="#">LDAR — 64-bit</a>	!
11	1	1	1	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CASA</a>	ARMv8.1
11	1	1	1	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CASAL</a>	ARMv8.1

**LoadLDAPR/storeSTLR exclusive(unscaled immediate)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	0	1	0	0	0	o2	L	o1				Rs		o0				Rt2						Rn					Rt	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	1	1	0	0	1	opc	0							imm9					0	0				Rn					Rt	

Decode fields					Rt2	Instruction Details	Architecture Version
size	o2	L	o1	o0			
	1		1		!= 11111	UNALLOCATED	!

size	o2	Decode fields			Rt2	Instruction Details	Architecture Version
		L	o1	o0			
0x	0		1		!= 11111	UNALLOCATED	-
00	0	0	0	0		STXRB	-
00	0	0	0	1		STLXRB	-
00	0	0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASP	ARMv8.1
00	0	0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPL	ARMv8.1
00	0	1	0	0		LDXRB	-
00	0	1	0	1		LDAXRB	-
00	0	1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPA	ARMv8.1
00	0	1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPAL	ARMv8.1
00	1	0	0	0		STLLRB	ARMv8.1
00	1	0	0	1		STLRB	-
00	1	0	1	0	11111	CASB, CASAB, CASALB, CASLB — CASB	ARMv8.1
00	1	0	1	1	11111	CASB, CASAB, CASALB, CASLB — CASLB	ARMv8.1
00	1	1	0	0		LDLARB	ARMv8.1
00	1	1	0	1		LDARB	-
00	1	1	1	0	11111	CASB, CASAB, CASALB, CASLB — CASAB	ARMv8.1
00	1	1	1	1	11111	CASB, CASAB, CASALB, CASLB — CASALB	ARMv8.1
01	0	0	0	0		STXRH	-
01	0	0	0	1		STLXRH	-
01	0	0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASP	ARMv8.1
01	0	0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPL	ARMv8.1
01	0	1	0	0		LDXRH	-
01	0	1	0	1		LDAXRH	-
01	0	1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPA	ARMv8.1
01	0	1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL	ARMv8.1
01	1	0	0	0		STLLRH	ARMv8.1
01	1	0	0	1		STLRH	-
01	1	0	1	0	11111	CASH, CASHA, CASALH, CASLH — CASH	ARMv8.1
01	1	0	1	1	11111	CASH, CASHA, CASALH, CASLH — CASLH	ARMv8.1
01	1	1	0	0		LDLARH	ARMv8.1
01	1	1	0	1		LDARH	-
01	1	1	1	0	11111	CASH, CASHA, CASALH, CASLH — CASHA	ARMv8.1
01	1	1	1	1	11111	CASH, CASHA, CASALH, CASLH — CASALH	ARMv8.1
10	0	0	0	0		STXR — 32-bit	-
10	0	0	0	1		STLXR — 32-bit	-
10	0	0	1	0		STXP — 32-bit	-
10	0	0	1	1		STLXP — 32-bit	-
10	0	1	0	0		LDXR — 32-bit	-
10	0	1	0	1		LDAXR — 32-bit	-
10	0	1	1	0		LDXP — 32-bit	-
10	0	1	1	1		LDAXP — 32-bit	-
10	1	0	0	0		STLLR — 32-bit	ARMv8.1
10	1	0	0	1		STLR — 32-bit	-
10	1	0	1	0	11111	CAS, CASA, CASAL, CASL — 32-bit CAS	ARMv8.1
10	1	0	1	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASL	ARMv8.1
10	1	1	0	0		LDLAR — 32-bit	ARMv8.1
10	1	1	0	1		LDAR — 32-bit	-

Decode fields					Rt2	Instruction Details	Architecture Version
size	o2	L	o1	o0			
10	1	1	1	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 32-bit CASA</a>	ARMv8.1
10	1	1	1	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 32-bit CASAL</a>	ARMv8.1
11	0	0	0	0		<a href="#">STXR — 64-bit</a>	-
11	0	0	0	1		<a href="#">STLXR — 64-bit</a>	-
11	0	0	1	0		<a href="#">STXP — 64-bit</a>	-
11	0	0	1	1		<a href="#">STLXP — 64-bit</a>	-
11	0	1	0	0		<a href="#">LDXR — 64-bit</a>	-
11	0	1	0	1		<a href="#">LDAXR — 64-bit</a>	-
11	0	1	1	0		<a href="#">LDXP — 64-bit</a>	-
11	0	1	1	1		<a href="#">LDAXP — 64-bit</a>	-
11	1	0	0	0		<a href="#">STLLR — 64-bit</a>	ARMv8.1
11	1	0	0	1		<a href="#">STLR — 64-bit</a>	-
11	1	0	1	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CAS</a>	ARMv8.1
11	1	0	1	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CASAL</a>	ARMv8.1
11	1	1	0	0		<a href="#">LDLAR — 64-bit</a>	ARMv8.1
11	1	1	0	1		<a href="#">LDAR — 64-bit</a>	-
11	1	1	1	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CASA</a>	ARMv8.1
11	1	1	1	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CASAL</a>	ARMv8.1

Decode fields		Instruction Details
size	opc	
00	00	<a href="#">STLURB</a>
00	01	<a href="#">LDAPURB</a>
00	10	<a href="#">LDAPURSB — 64-bit</a>
00	11	<a href="#">LDAPURSB — 32-bit</a>
01	00	<a href="#">STLURH</a>
01	01	<a href="#">LDAPURH</a>
01	10	<a href="#">LDAPURSH — 64-bit</a>
01	11	<a href="#">LDAPURSH — 32-bit</a>
10	00	<a href="#">STLUR — 32-bit</a>
10	01	<a href="#">LDAPUR — 32-bit</a>
10	10	<a href="#">LDAPURSW</a>
10	11	UNALLOCATED
11	00	<a href="#">STLUR — 64-bit</a>
11	01	<a href="#">LDAPUR — 64-bit</a>
11	10	UNALLOCATED
11	11	UNALLOCATED

## Load register (literal)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		0	1	1	V	0	0	imm19																		Rt					

Decode fields		Instruction Details
opc	V	
00	0	LDR (literal) — 32-bit
00	1	<a href="#">LDR (literal, SIMD&amp;FP) — 32-bit</a>
01	0	LDR (literal) — 64-bit

Decode fields opc	V	Instruction Details
01	1	<a href="#">LDR (literal, SIMD&amp;FP)</a> — <a href="#">64-bit</a>
10	0	LDRSW (literal)
10	1	<a href="#">LDR (literal, SIMD&amp;FP)</a> — <a href="#">128-bit</a>
11	0	PRFM (literal)
11	1	UNALLOCATED

**Load/store no-allocate pair (offset)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	V	0	0	0	L	imm7							Rt2					Rn					Rt				

Decode fields opc	V	L	Instruction Details
00	0	0	<a href="#">STNP</a> — <a href="#">32-bit</a>
00	0	1	<a href="#">LDNP</a> — <a href="#">32-bit</a>
00	1	0	<a href="#">STNP (SIMD&amp;FP)</a> — <a href="#">32-bit</a>
00	1	1	<a href="#">LDNP (SIMD&amp;FP)</a> — <a href="#">32-bit</a>
01	0		UNALLOCATED
01	1	0	<a href="#">STNP (SIMD&amp;FP)</a> — <a href="#">64-bit</a>
01	1	1	<a href="#">LDNP (SIMD&amp;FP)</a> — <a href="#">64-bit</a>
10	0	0	<a href="#">STNP</a> — <a href="#">64-bit</a>
10	0	1	<a href="#">LDNP</a> — <a href="#">64-bit</a>
10	1	0	<a href="#">STNP (SIMD&amp;FP)</a> — <a href="#">128-bit</a>
10	1	1	<a href="#">LDNP (SIMD&amp;FP)</a> — <a href="#">128-bit</a>
11			UNALLOCATED

**Load/store register pair (post-indexed)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	V	0	0	1	L	imm7							Rt2					Rn					Rt				

Decode fields opc	V	L	Instruction Details	Architecture Version
00	0	0	<a href="#">STP</a> — <a href="#">32-bit</a>	⌋
00	0	1	<a href="#">LDP</a> — <a href="#">32-bit</a>	⌋
00	1	0	<a href="#">STP (SIMD&amp;FP)</a> — <a href="#">32-bit</a>	⌋
00	1	1	<a href="#">LDP (SIMD&amp;FP)</a> — <a href="#">32-bit</a>	⌋
01	0	0	UNALLOCATEDSTGP	ARMv8.5
01	0	1	<a href="#">LDPSW</a>	⌋
01	1	0	<a href="#">STP (SIMD&amp;FP)</a> — <a href="#">64-bit</a>	⌋
01	1	1	<a href="#">LDP (SIMD&amp;FP)</a> — <a href="#">64-bit</a>	⌋
10	0	0	<a href="#">STP</a> — <a href="#">64-bit</a>	⌋
10	0	1	<a href="#">LDP</a> — <a href="#">64-bit</a>	⌋
10	1	0	<a href="#">STP (SIMD&amp;FP)</a> — <a href="#">128-bit</a>	⌋
10	1	1	<a href="#">LDP (SIMD&amp;FP)</a> — <a href="#">128-bit</a>	⌋
11			UNALLOCATED	⌋

**Load/store register pair (offset)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc	1	0	1	V	0	1	0	L	imm7							Rt2				Rn				Rt							

Decode fields			Instruction Details	Architecture Version
opc	V	L		
00	0	0	<a href="#">STP</a> — <a href="#">32-bit</a>	—
00	0	1	<a href="#">LDP</a> — <a href="#">32-bit</a>	—
00	1	0	<a href="#">STP (SIMD&amp;FP)</a> — <a href="#">32-bit</a>	—
00	1	1	<a href="#">LDP (SIMD&amp;FP)</a> — <a href="#">32-bit</a>	—
01	0	0	UNALLOCATEDSTGP	ARMv8.5
01	0	1	<a href="#">LDPSW</a>	—
01	1	0	<a href="#">STP (SIMD&amp;FP)</a> — <a href="#">64-bit</a>	—
01	1	1	<a href="#">LDP (SIMD&amp;FP)</a> — <a href="#">64-bit</a>	—
10	0	0	<a href="#">STP</a> — <a href="#">64-bit</a>	—
10	0	1	<a href="#">LDP</a> — <a href="#">64-bit</a>	—
10	1	0	<a href="#">STP (SIMD&amp;FP)</a> — <a href="#">128-bit</a>	—
10	1	1	<a href="#">LDP (SIMD&amp;FP)</a> — <a href="#">128-bit</a>	—
11			UNALLOCATED	—

**Load/store register pair (pre-indexed)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc	1	0	1	V	0	1	1	L	imm7							Rt2				Rn				Rt							

Decode fields			Instruction Details	Architecture Version
opc	V	L		
00	0	0	<a href="#">STP</a> — <a href="#">32-bit</a>	—
00	0	1	<a href="#">LDP</a> — <a href="#">32-bit</a>	—
00	1	0	<a href="#">STP (SIMD&amp;FP)</a> — <a href="#">32-bit</a>	—
00	1	1	<a href="#">LDP (SIMD&amp;FP)</a> — <a href="#">32-bit</a>	—
01	0	0	UNALLOCATEDSTGP	ARMv8.5
01	0	1	<a href="#">LDPSW</a>	—
01	1	0	<a href="#">STP (SIMD&amp;FP)</a> — <a href="#">64-bit</a>	—
01	1	1	<a href="#">LDP (SIMD&amp;FP)</a> — <a href="#">64-bit</a>	—
10	0	0	<a href="#">STP</a> — <a href="#">64-bit</a>	—
10	0	1	<a href="#">LDP</a> — <a href="#">64-bit</a>	—
10	1	0	<a href="#">STP (SIMD&amp;FP)</a> — <a href="#">128-bit</a>	—
10	1	1	<a href="#">LDP (SIMD&amp;FP)</a> — <a href="#">128-bit</a>	—
11			UNALLOCATED	—

**Load/store register (unscaled immediate)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	V	0	0	opc	0	imm9							0 0				Rn				Rt							



Decode fields			Instruction Details
size	V	opc	
×1	1	1×	UNALLOCATED
00	0	00	<a href="#">STURB</a>
00	0	01	<a href="#">LDURB</a>
00	0	10	<a href="#">LDURSB</a> — 64-bit
00	0	11	<a href="#">LDURSB</a> — 32-bit
00	1	00	<a href="#">STUR (SIMD&amp;FP)</a> — 8-bit
00	1	01	<a href="#">LDUR (SIMD&amp;FP)</a> — 8-bit
00	1	10	<a href="#">STUR (SIMD&amp;FP)</a> — 128-bit
00	1	11	<a href="#">LDUR (SIMD&amp;FP)</a> — 128-bit
01	0	00	<a href="#">STURH</a>
01	0	01	<a href="#">LDURH</a>
01	0	10	<a href="#">LDURSH</a> — 64-bit
01	0	11	<a href="#">LDURSH</a> — 32-bit
01	1	00	<a href="#">STUR (SIMD&amp;FP)</a> — 16-bit
01	1	01	<a href="#">LDUR (SIMD&amp;FP)</a> — 16-bit
1×	0	11	UNALLOCATED
1×	1	1×	UNALLOCATED
10	0	00	<a href="#">STUR</a> — 32-bit
10	0	01	<a href="#">LDUR</a> — 32-bit
10	0	10	<a href="#">LDURSW</a>
10	1	00	<a href="#">STUR (SIMD&amp;FP)</a> — 32-bit
10	1	01	<a href="#">LDUR (SIMD&amp;FP)</a> — 32-bit
11	0	00	<a href="#">STUR</a> — 64-bit
11	0	01	<a href="#">LDUR</a> — 64-bit
11	0	10	<a href="#">PRFM (unscaled offset)</a>
11	1	00	<a href="#">STUR (SIMD&amp;FP)</a> — 64-bit
11	1	01	<a href="#">LDUR (SIMD&amp;FP)</a> — 64-bit

**Load/store register (immediate post-indexed)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	0	opc		0	imm9										0	1	Rn				Rt				

Decode fields			Instruction Details
size	V	opc	
×1	1	1×	UNALLOCATED
00	0	00	<a href="#">STRB (immediate)</a>
00	0	01	<a href="#">LDRB (immediate)</a>
00	0	10	<a href="#">LDRSB (immediate)</a> — 64-bit
00	0	11	<a href="#">LDRSB (immediate)</a> — 32-bit
00	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 8-bit
00	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 8-bit
00	1	10	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 128-bit
00	1	11	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 128-bit
01	0	00	<a href="#">STRH (immediate)</a>
01	0	01	<a href="#">LDRH (immediate)</a>
01	0	10	<a href="#">LDRSH (immediate)</a> — 64-bit

Decode fields			Instruction Details
size	V	opc	
01	0	11	<a href="#">LDRSH (immediate)</a> — <a href="#">32-bit</a>
01	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — <a href="#">16-bit</a>
01	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — <a href="#">16-bit</a>
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	<a href="#">STR (immediate)</a> — <a href="#">32-bit</a>
10	0	01	<a href="#">LDR (immediate)</a> — <a href="#">32-bit</a>
10	0	10	<a href="#">LDRSW (immediate)</a>
10	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — <a href="#">32-bit</a>
10	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — <a href="#">32-bit</a>
11	0	00	<a href="#">STR (immediate)</a> — <a href="#">64-bit</a>
11	0	01	<a href="#">LDR (immediate)</a> — <a href="#">64-bit</a>
11	0	10	UNALLOCATED
11	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — <a href="#">64-bit</a>
11	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — <a href="#">64-bit</a>

**Load/store register (unprivileged)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	0	opc		0	imm9									1	0	Rn					Rt				

Decode fields			Instruction Details
size	V	opc	
	1		UNALLOCATED
00	0	00	<a href="#">STTRB</a>
00	0	01	<a href="#">LDTRB</a>
00	0	10	<a href="#">LDTRSB</a> — <a href="#">64-bit</a>
00	0	11	<a href="#">LDTRSB</a> — <a href="#">32-bit</a>
01	0	00	<a href="#">STTRH</a>
01	0	01	<a href="#">LDTRH</a>
01	0	10	<a href="#">LDTRSH</a> — <a href="#">64-bit</a>
01	0	11	<a href="#">LDTRSH</a> — <a href="#">32-bit</a>
1x	0	11	UNALLOCATED
10	0	00	<a href="#">STTR</a> — <a href="#">32-bit</a>
10	0	01	<a href="#">LDTR</a> — <a href="#">32-bit</a>
10	0	10	<a href="#">LDTRSW</a>
11	0	00	<a href="#">STTR</a> — <a href="#">64-bit</a>
11	0	01	<a href="#">LDTR</a> — <a href="#">64-bit</a>
11	0	10	UNALLOCATED

**Load/store register (immediate pre-indexed)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	0	opc		0	imm9										1	1	Rn					Rt			

Decode fields			Instruction Details
size	V	opc	
×1	1	1×	UNALLOCATED
00	0	00	<a href="#">STRB (immediate)</a>
00	0	01	<a href="#">LDRB (immediate)</a>
00	0	10	<a href="#">LDRSB (immediate)</a> — <a href="#">64-bit</a>
00	0	11	<a href="#">LDRSB (immediate)</a> — <a href="#">32-bit</a>
00	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — <a href="#">8-bit</a>
00	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — <a href="#">8-bit</a>
00	1	10	<a href="#">STR (immediate, SIMD&amp;FP)</a> — <a href="#">128-bit</a>
00	1	11	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — <a href="#">128-bit</a>
01	0	00	<a href="#">STRH (immediate)</a>
01	0	01	<a href="#">LDRH (immediate)</a>
01	0	10	<a href="#">LDRSH (immediate)</a> — <a href="#">64-bit</a>
01	0	11	<a href="#">LDRSH (immediate)</a> — <a href="#">32-bit</a>
01	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — <a href="#">16-bit</a>
01	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — <a href="#">16-bit</a>
1×	0	11	UNALLOCATED
1×	1	1×	UNALLOCATED
10	0	00	<a href="#">STR (immediate)</a> — <a href="#">32-bit</a>
10	0	01	<a href="#">LDR (immediate)</a> — <a href="#">32-bit</a>
10	0	10	<a href="#">LDRSW (immediate)</a>
10	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — <a href="#">32-bit</a>
10	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — <a href="#">32-bit</a>
11	0	00	<a href="#">STR (immediate)</a> — <a href="#">64-bit</a>
11	0	01	<a href="#">LDR (immediate)</a> — <a href="#">64-bit</a>
11	0	10	UNALLOCATED
11	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — <a href="#">64-bit</a>
11	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — <a href="#">64-bit</a>

## Atomic memory operations

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	0	A	R	1	Rs				o3	opc			0		0	Rn					Rt				

Decode fields						Instruction Details	Architecture Version
size	V	A	R	o3	opc		
	0			1	001	UNALLOCATED	-
	0			1	01×	UNALLOCATED	-
	0			1	101	UNALLOCATED	-
	0			1	11×	UNALLOCATED	-
	0	0		1	100	UNALLOCATED	-
	0	1	1	1	100	UNALLOCATED	-
	1					UNALLOCATED	-
00	0	0	0	0	000	<a href="#">LDADDB, LDADDAB, LDADDALB, LDADDLB</a> — <a href="#">LDADDB</a>	ARMv8.1
00	0	0	0	0	001	<a href="#">LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB</a> — <a href="#">LDCLRB</a>	ARMv8.1
00	0	0	0	0	010	<a href="#">LDEORB, LDEORAB, LDEORALB, LDEORLB</a> — <a href="#">LDEORB</a>	ARMv8.1
00	0	0	0	0	011	<a href="#">LDSETB, LDSETAB, LDSETALB, LDSETLB</a> — <a href="#">LDSETB</a>	ARMv8.1

size	Decode fields					Instruction Details	Architecture Version
	V	A	R	o3	opc		
00	0	0	0	0	100	<a href="#">LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB</a> — <a href="#">LDSMAXB</a>	ARMv8.1
00	0	0	0	0	101	<a href="#">LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB</a> — <a href="#">LDSMINB</a>	ARMv8.1
00	0	0	0	0	110	<a href="#">LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB</a> — <a href="#">LDUMAXB</a>	ARMv8.1
00	0	0	0	0	111	<a href="#">LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB</a> — <a href="#">LDUMINB</a>	ARMv8.1
00	0	0	0	1	000	<a href="#">SWPB, SWPAB, SWPALB, SWPLB</a> — <a href="#">SWPB</a>	ARMv8.1
00	0	0	1	0	000	<a href="#">LDADDB, LDADDAB, LDADDALB, LDADDLB</a> — <a href="#">LDADDLB</a>	ARMv8.1
00	0	0	1	0	001	<a href="#">LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB</a> — <a href="#">LDCLRLB</a>	ARMv8.1
00	0	0	1	0	010	<a href="#">LDEORB, LDEORAB, LDEORALB, LDEORLB</a> — <a href="#">LDEORLB</a>	ARMv8.1
00	0	0	1	0	011	<a href="#">LDSETB, LDSETAB, LDSETALB, LDSETLB</a> — <a href="#">LDSETLB</a>	ARMv8.1
00	0	0	1	0	100	<a href="#">LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB</a> — <a href="#">LDSMAXLB</a>	ARMv8.1
00	0	0	1	0	101	<a href="#">LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB</a> — <a href="#">LDSMINLB</a>	ARMv8.1
00	0	0	1	0	110	<a href="#">LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB</a> — <a href="#">LDUMAXLB</a>	ARMv8.1
00	0	0	1	0	111	<a href="#">LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB</a> — <a href="#">LDUMINLB</a>	ARMv8.1
00	0	0	1	1	000	<a href="#">SWPB, SWPAB, SWPALB, SWPLB</a> — <a href="#">SWPLB</a>	ARMv8.1
00	0	1	0	0	000	<a href="#">LDADDB, LDADDAB, LDADDALB, LDADDLB</a> — <a href="#">LDADDAB</a>	ARMv8.1
00	0	1	0	0	001	<a href="#">LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB</a> — <a href="#">LDCLRAB</a>	ARMv8.1
00	0	1	0	0	010	<a href="#">LDEORB, LDEORAB, LDEORALB, LDEORLB</a> — <a href="#">LDEORAB</a>	ARMv8.1
00	0	1	0	0	011	<a href="#">LDSETB, LDSETAB, LDSETALB, LDSETLB</a> — <a href="#">LDSETAB</a>	ARMv8.1
00	0	1	0	0	100	<a href="#">LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB</a> — <a href="#">LDSMAXAB</a>	ARMv8.1
00	0	1	0	0	101	<a href="#">LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB</a> — <a href="#">LDSMINAB</a>	ARMv8.1
00	0	1	0	0	110	<a href="#">LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB</a> — <a href="#">LDUMAXAB</a>	ARMv8.1
00	0	1	0	0	111	<a href="#">LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB</a> — <a href="#">LDUMINAB</a>	ARMv8.1
00	0	1	0	1	000	<a href="#">SWPB, SWPAB, SWPALB, SWPLB</a> — <a href="#">SWPAB</a>	ARMv8.1
00	0	1	0	1	100	<a href="#">LDAPRB</a>	ARMv8.3
00	0	1	1	0	000	<a href="#">LDADDB, LDADDAB, LDADDALB, LDADDLB</a> — <a href="#">LDADDALB</a>	ARMv8.1
00	0	1	1	0	001	<a href="#">LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB</a> — <a href="#">LDCLRALB</a>	ARMv8.1
00	0	1	1	0	010	<a href="#">LDEORB, LDEORAB, LDEORALB, LDEORLB</a> — <a href="#">LDEORALB</a>	ARMv8.1
00	0	1	1	0	011	<a href="#">LDSETB, LDSETAB, LDSETALB, LDSETLB</a> — <a href="#">LDSETALB</a>	ARMv8.1
00	0	1	1	0	100	<a href="#">LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB</a> — <a href="#">LDSMAXALB</a>	ARMv8.1
00	0	1	1	0	101	<a href="#">LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB</a> — <a href="#">LDSMINALB</a>	ARMv8.1
00	0	1	1	0	110	<a href="#">LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB</a> — <a href="#">LDUMAXALB</a>	ARMv8.1
00	0	1	1	0	111	<a href="#">LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB</a> — <a href="#">LDUMINALB</a>	ARMv8.1
00	0	1	1	1	000	<a href="#">SWPB, SWPAB, SWPALB, SWPLB</a> — <a href="#">SWPALB</a>	ARMv8.1
01	0	0	0	0	000	<a href="#">LDADDB, LDADDAB, LDADDALB, LDADDLB</a> — <a href="#">LDADDBH</a>	ARMv8.1
01	0	0	0	0	001	<a href="#">LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH</a> — <a href="#">LDCLRH</a>	ARMv8.1
01	0	0	0	0	010	<a href="#">LDEORH, LDEORAH, LDEORALH, LDEORLH</a> — <a href="#">LDEORH</a>	ARMv8.1
01	0	0	0	0	011	<a href="#">LDSETH, LDSETAH, LDSETALH, LDSETLH</a> — <a href="#">LDSETH</a>	ARMv8.1
01	0	0	0	0	100	<a href="#">LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH</a> — <a href="#">LDSMAXH</a>	ARMv8.1
01	0	0	0	0	101	<a href="#">LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH</a> — <a href="#">LDSMINH</a>	ARMv8.1
01	0	0	0	0	110	<a href="#">LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH</a> — <a href="#">LDUMAXH</a>	ARMv8.1

size	Decode fields					Instruction Details	Architecture Version
	V	A	R	o3	opc		
01	0	0	0	0	111	<a href="#">LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH</a> — <a href="#">LDUMINH</a>	ARMv8.1
01	0	0	0	1	000	<a href="#">SWPH, SWPAH, SWPALH, SWPLH</a> — <a href="#">SWPH</a>	ARMv8.1
01	0	0	1	0	000	<a href="#">LDADDH, LDADDAH, LDADDALH, LDADDLH</a> — <a href="#">LDADDLH</a>	ARMv8.1
01	0	0	1	0	001	<a href="#">LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH</a> — <a href="#">LDCLRLH</a>	ARMv8.1
01	0	0	1	0	010	<a href="#">LDEORH, LDEORAH, LDEORALH, LDEORLH</a> — <a href="#">LDEORLH</a>	ARMv8.1
01	0	0	1	0	011	<a href="#">LDSETH, LDSETAH, LDSETALH, LDSETLH</a> — <a href="#">LDSETLH</a>	ARMv8.1
01	0	0	1	0	100	<a href="#">LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH</a> — <a href="#">LDSMAXLH</a>	ARMv8.1
01	0	0	1	0	101	<a href="#">LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH</a> — <a href="#">LDSMINLH</a>	ARMv8.1
01	0	0	1	0	110	<a href="#">LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH</a> — <a href="#">LDUMAXLH</a>	ARMv8.1
01	0	0	1	0	111	<a href="#">LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH</a> — <a href="#">LDUMINLH</a>	ARMv8.1
01	0	0	1	1	000	<a href="#">SWPH, SWPAH, SWPALH, SWPLH</a> — <a href="#">SWPLH</a>	ARMv8.1
01	0	1	0	0	000	<a href="#">LDADDH, LDADDAH, LDADDALH, LDADDLH</a> — <a href="#">LDADDAH</a>	ARMv8.1
01	0	1	0	0	001	<a href="#">LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH</a> — <a href="#">LDCLRAH</a>	ARMv8.1
01	0	1	0	0	010	<a href="#">LDEORH, LDEORAH, LDEORALH, LDEORLH</a> — <a href="#">LDEORAH</a>	ARMv8.1
01	0	1	0	0	011	<a href="#">LDSETH, LDSETAH, LDSETALH, LDSETLH</a> — <a href="#">LDSETAH</a>	ARMv8.1
01	0	1	0	0	100	<a href="#">LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH</a> — <a href="#">LDSMAXAH</a>	ARMv8.1
01	0	1	0	0	101	<a href="#">LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH</a> — <a href="#">LDSMINAH</a>	ARMv8.1
01	0	1	0	0	110	<a href="#">LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH</a> — <a href="#">LDUMAXAH</a>	ARMv8.1
01	0	1	0	0	111	<a href="#">LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH</a> — <a href="#">LDUMINAH</a>	ARMv8.1
01	0	1	0	1	000	<a href="#">SWPH, SWPAH, SWPALH, SWPLH</a> — <a href="#">SWPAH</a>	ARMv8.1
01	0	1	0	1	100	<a href="#">LDAPRH</a>	ARMv8.3
01	0	1	1	0	000	<a href="#">LDADDH, LDADDAH, LDADDALH, LDADDLH</a> — <a href="#">LDADDALH</a>	ARMv8.1
01	0	1	1	0	001	<a href="#">LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH</a> — <a href="#">LDCLRALH</a>	ARMv8.1
01	0	1	1	0	010	<a href="#">LDEORH, LDEORAH, LDEORALH, LDEORLH</a> — <a href="#">LDEORALH</a>	ARMv8.1
01	0	1	1	0	011	<a href="#">LDSETH, LDSETAH, LDSETALH, LDSETLH</a> — <a href="#">LDSETALH</a>	ARMv8.1
01	0	1	1	0	100	<a href="#">LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH</a> — <a href="#">LDSMAXALH</a>	ARMv8.1
01	0	1	1	0	101	<a href="#">LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH</a> — <a href="#">LDSMINALH</a>	ARMv8.1
01	0	1	1	0	110	<a href="#">LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH</a> — <a href="#">LDUMAXALH</a>	ARMv8.1
01	0	1	1	0	111	<a href="#">LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH</a> — <a href="#">LDUMINALH</a>	ARMv8.1
01	0	1	1	1	000	<a href="#">SWPH, SWPAH, SWPALH, SWPLH</a> — <a href="#">SWPALH</a>	ARMv8.1
10	0	0	0	0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — <a href="#">32-bit LDADD</a>	ARMv8.1
10	0	0	0	0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — <a href="#">32-bit LDCLR</a>	ARMv8.1
10	0	0	0	0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — <a href="#">32-bit LDEOR</a>	ARMv8.1
10	0	0	0	0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — <a href="#">32-bit LDSET</a>	ARMv8.1
10	0	0	0	0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — <a href="#">32-bit LDSMAX</a>	ARMv8.1
10	0	0	0	0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — <a href="#">32-bit LDSMIN</a>	ARMv8.1
10	0	0	0	0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — <a href="#">32-bit LDUMAX</a>	ARMv8.1
10	0	0	0	0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — <a href="#">32-bit LDUMIN</a>	ARMv8.1
10	0	0	0	1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — <a href="#">32-bit SWP</a>	ARMv8.1
10	0	0	1	0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — <a href="#">32-bit LDADDL</a>	ARMv8.1
10	0	0	1	0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — <a href="#">32-bit LDCLRL</a>	ARMv8.1
10	0	0	1	0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — <a href="#">32-bit LDEORL</a>	ARMv8.1

size	Decode fields					Instruction Details	Architecture Version
	V	A	R	o3	opc		
10	0	0	1	0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 32-bit <a href="#">LDSETL</a>	ARMv8.1
10	0	0	1	0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — 32-bit <a href="#">LDSMAXL</a>	ARMv8.1
10	0	0	1	0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — 32-bit <a href="#">LDSMINL</a>	ARMv8.1
10	0	0	1	0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — 32-bit <a href="#">LDUMAXL</a>	ARMv8.1
10	0	0	1	0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — 32-bit <a href="#">LDUMINL</a>	ARMv8.1
10	0	0	1	1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 32-bit <a href="#">SWPL</a>	ARMv8.1
10	0	1	0	0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 32-bit <a href="#">LDADDA</a>	ARMv8.1
10	0	1	0	0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 32-bit <a href="#">LDCLRA</a>	ARMv8.1
10	0	1	0	0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 32-bit <a href="#">LDEORA</a>	ARMv8.1
10	0	1	0	0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 32-bit <a href="#">LDSETA</a>	ARMv8.1
10	0	1	0	0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — 32-bit <a href="#">LDSMAXA</a>	ARMv8.1
10	0	1	0	0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — 32-bit <a href="#">LDSMINA</a>	ARMv8.1
10	0	1	0	0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — 32-bit <a href="#">LDUMAXA</a>	ARMv8.1
10	0	1	0	0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — 32-bit <a href="#">LDUMINA</a>	ARMv8.1
10	0	1	0	1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 32-bit <a href="#">SWPA</a>	ARMv8.1
10	0	1	0	1	100	<a href="#">LDAPR</a> — 32-bit	ARMv8.3
10	0	1	1	0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 32-bit <a href="#">LDADDAL</a>	ARMv8.1
10	0	1	1	0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 32-bit <a href="#">LDCLRAL</a>	ARMv8.1
10	0	1	1	0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 32-bit <a href="#">LDEORAL</a>	ARMv8.1
10	0	1	1	0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 32-bit <a href="#">LDSETAL</a>	ARMv8.1
10	0	1	1	0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — 32-bit <a href="#">LDSMAXAL</a>	ARMv8.1
10	0	1	1	0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — 32-bit <a href="#">LDSMINAL</a>	ARMv8.1
10	0	1	1	0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — 32-bit <a href="#">LDUMAXAL</a>	ARMv8.1
10	0	1	1	0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — 32-bit <a href="#">LDUMINAL</a>	ARMv8.1
10	0	1	1	1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 32-bit <a href="#">SWPAL</a>	ARMv8.1
11	0	0	0	0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 64-bit <a href="#">LDADD</a>	ARMv8.1
11	0	0	0	0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 64-bit <a href="#">LDCLR</a>	ARMv8.1
11	0	0	0	0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 64-bit <a href="#">LDEOR</a>	ARMv8.1
11	0	0	0	0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 64-bit <a href="#">LDSET</a>	ARMv8.1
11	0	0	0	0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — 64-bit <a href="#">LDSMAX</a>	ARMv8.1
11	0	0	0	0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — 64-bit <a href="#">LDSMIN</a>	ARMv8.1
11	0	0	0	0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — 64-bit <a href="#">LDUMAX</a>	ARMv8.1
11	0	0	0	0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — 64-bit <a href="#">LDUMIN</a>	ARMv8.1
11	0	0	0	1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 64-bit <a href="#">SWP</a>	ARMv8.1
11	0	0	1	0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 64-bit <a href="#">LDADDL</a>	ARMv8.1
11	0	0	1	0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 64-bit <a href="#">LDCLRL</a>	ARMv8.1
11	0	0	1	0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 64-bit <a href="#">LDEORL</a>	ARMv8.1
11	0	0	1	0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 64-bit <a href="#">LDSETL</a>	ARMv8.1
11	0	0	1	0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — 64-bit <a href="#">LDSMAXL</a>	ARMv8.1
11	0	0	1	0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — 64-bit <a href="#">LDSMINL</a>	ARMv8.1
11	0	0	1	0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — 64-bit <a href="#">LDUMAXL</a>	ARMv8.1
11	0	0	1	0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — 64-bit <a href="#">LDUMINL</a>	ARMv8.1

size	Decode fields					Instruction Details	Architecture Version
	V	A	R	o3	opc		
11	0	0	1	1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — <a href="#">64-bit SWPL</a>	ARMv8.1
11	0	1	0	0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — <a href="#">64-bit LDADDA</a>	ARMv8.1
11	0	1	0	0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — <a href="#">64-bit LDCLRA</a>	ARMv8.1
11	0	1	0	0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — <a href="#">64-bit LDEORA</a>	ARMv8.1
11	0	1	0	0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — <a href="#">64-bit LDSETA</a>	ARMv8.1
11	0	1	0	0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — <a href="#">64-bit LDSMAXA</a>	ARMv8.1
11	0	1	0	0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — <a href="#">64-bit LDSMINA</a>	ARMv8.1
11	0	1	0	0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — <a href="#">64-bit LDUMAXA</a>	ARMv8.1
11	0	1	0	0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — <a href="#">64-bit LDUMINA</a>	ARMv8.1
11	0	1	0	1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — <a href="#">64-bit SWPA</a>	ARMv8.1
11	0	1	0	1	100	<a href="#">LDAPR</a> — <a href="#">64-bit</a>	ARMv8.3
11	0	1	1	0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — <a href="#">64-bit LDADDAL</a>	ARMv8.1
11	0	1	1	0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — <a href="#">64-bit LDCLRAL</a>	ARMv8.1
11	0	1	1	0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — <a href="#">64-bit LDEORAL</a>	ARMv8.1
11	0	1	1	0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — <a href="#">64-bit LDSETAL</a>	ARMv8.1
11	0	1	1	0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — <a href="#">64-bit LDSMAXAL</a>	ARMv8.1
11	0	1	1	0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — <a href="#">64-bit LDSMINAL</a>	ARMv8.1
11	0	1	1	0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — <a href="#">64-bit LDUMAXAL</a>	ARMv8.1
11	0	1	1	0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — <a href="#">64-bit LDUMINAL</a>	ARMv8.1
11	0	1	1	1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — <a href="#">64-bit SWPAL</a>	ARMv8.1

**Load/store register (register offset)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	0	opc		1	Rm				option			S	1	0	Rn				Rt						

size	Decode fields			Instruction Details
	V	opc	option	
			x0x	UNALLOCATED
x1	1	1x		UNALLOCATED
00	0	00	!= 011	<a href="#">STRB (register)</a> — <a href="#">extended register</a>
00	0	00	011	<a href="#">STRB (register)</a> — <a href="#">shifted register</a>
00	0	01	!= 011	<a href="#">LDRB (register)</a> — <a href="#">extended register</a>
00	0	01	011	<a href="#">LDRB (register)</a> — <a href="#">shifted register</a>
00	0	10	!= 011	<a href="#">LDRSB (register)</a> — <a href="#">64-bit with extended register offset</a>
00	0	10	011	<a href="#">LDRSB (register)</a> — <a href="#">64-bit with shifted register offset</a>
00	0	11	!= 011	<a href="#">LDRSB (register)</a> — <a href="#">32-bit with extended register offset</a>
00	0	11	011	<a href="#">LDRSB (register)</a> — <a href="#">32-bit with shifted register offset</a>
00	1	00	!= 011	<a href="#">STR (register, SIMD&amp;FP)</a>
00	1	00	011	<a href="#">STR (register, SIMD&amp;FP)</a>
00	1	01	!= 011	<a href="#">LDR (register, SIMD&amp;FP)</a>
00	1	01	011	<a href="#">LDR (register, SIMD&amp;FP)</a>
00	1	10		<a href="#">STR (register, SIMD&amp;FP)</a>
00	1	11		<a href="#">LDR (register, SIMD&amp;FP)</a>

Decode fields				Instruction Details
size	V	opc	option	
01	0	00		<a href="#">STRH (register)</a>
01	0	01		<a href="#">LDRH (register)</a>
01	0	10		<a href="#">LDRSH (register)</a> — 64-bit
01	0	11		<a href="#">LDRSH (register)</a> — 32-bit
01	1	00		<a href="#">STR (register, SIMD&amp;FP)</a>
01	1	01		<a href="#">LDR (register, SIMD&amp;FP)</a>
1x	0	11		UNALLOCATED
1x	1	1x		UNALLOCATED
10	0	00		<a href="#">STR (register)</a> — 32-bit
10	0	01		<a href="#">LDR (register)</a> — 32-bit
10	0	10		<a href="#">LDRSW (register)</a>
10	1	00		<a href="#">STR (register, SIMD&amp;FP)</a>
10	1	01		<a href="#">LDR (register, SIMD&amp;FP)</a>
11	0	00		<a href="#">STR (register)</a> — 64-bit
11	0	01		<a href="#">LDR (register)</a> — 64-bit
11	0	10		<a href="#">PRFM (register)</a>
11	1	00		<a href="#">STR (register, SIMD&amp;FP)</a>
11	1	01		<a href="#">LDR (register, SIMD&amp;FP)</a>

**Load/store register (pac)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	V	0	0	M	S	1	imm9											W	1	Rn				Rt				

Decode fields			Instruction Details	Architecture Version
size	M	W		
!= 11			UNALLOCATED	-
11	0	0	<a href="#">LDRAA, LDRAB</a> — <a href="#">key A, offset</a>	ARMv8.3
11	0	1	<a href="#">LDRAA, LDRAB</a> — <a href="#">key A, pre-indexed</a>	ARMv8.3
11	1	0	<a href="#">LDRAA, LDRAB</a> — <a href="#">key B, offset</a>	ARMv8.3
11	1	1	<a href="#">LDRAA, LDRAB</a> — <a href="#">key B, pre-indexed</a>	ARMv8.3

**Load/store register (unsigned immediate)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	1	opc		imm12											Rn				Rt						

Decode fields			Instruction Details
size	V	opc	
x1	1	1x	UNALLOCATED
00	0	00	<a href="#">STRB (immediate)</a>
00	0	01	<a href="#">LDRB (immediate)</a>
00	0	10	<a href="#">LDRSB (immediate)</a> — 64-bit
00	0	11	<a href="#">LDRSB (immediate)</a> — 32-bit
00	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 8-bit
00	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 8-bit



Decode fields			Instruction Details
size	V	opc	
00	1	10	<a href="#">STR (immediate, SIMD&amp;FP) — 128-bit</a>
00	1	11	<a href="#">LDR (immediate, SIMD&amp;FP) — 128-bit</a>
01	0	00	<a href="#">STRH (immediate)</a>
01	0	01	<a href="#">LDRH (immediate)</a>
01	0	10	<a href="#">LDRSH (immediate) — 64-bit</a>
01	0	11	<a href="#">LDRSH (immediate) — 32-bit</a>
01	1	00	<a href="#">STR (immediate, SIMD&amp;FP) — 16-bit</a>
01	1	01	<a href="#">LDR (immediate, SIMD&amp;FP) — 16-bit</a>
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	<a href="#">STR (immediate) — 32-bit</a>
10	0	01	<a href="#">LDR (immediate) — 32-bit</a>
10	0	10	<a href="#">LDRSW (immediate)</a>
10	1	00	<a href="#">STR (immediate, SIMD&amp;FP) — 32-bit</a>
10	1	01	<a href="#">LDR (immediate, SIMD&amp;FP) — 32-bit</a>
11	0	00	<a href="#">STR (immediate) — 64-bit</a>
11	0	01	<a href="#">LDR (immediate) — 64-bit</a>
11	0	10	<a href="#">PRFM (immediate)</a>
11	1	00	<a href="#">STR (immediate, SIMD&amp;FP) — 64-bit</a>
11	1	01	<a href="#">LDR (immediate, SIMD&amp;FP) — 64-bit</a>

## Data Processing -- Register

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0		op1		101		op2			op3																						

Decode fields			Instruction details	
op0	op1	op2	op3	
0	1	0110		<a href="#">Data-processing (2 source)</a>
1	1	0110		<a href="#">Data-processing (1 source)</a>
	0	0xxx		<a href="#">Logical (shifted register)</a>
	0	1xx0		<a href="#">Add/subtract (shifted register)</a>
	0	1xx1		<a href="#">Add/subtract (extended register)</a>
	1	0000	000000	<a href="#">Add/subtract (with carry)</a>
	1	0000	000011	UNALLOCATED
	1	0000	0001xx	UNALLOCATED
	1	0000	001xxx	UNALLOCATED
	1	0000	01xxxx	UNALLOCATED
	1	0000	1xxxxx	UNALLOCATED
	1	0000	x00001	<a href="#">Rotate right into flags</a>
	1	0000	xx0010	<a href="#">Evaluate into flags</a>
	1	0010	xxxx0x	<a href="#">Conditional compare (register)</a>
	1	0010	xxxx1x	<a href="#">Conditional compare (immediate)</a>
	1	0100		<a href="#">Conditional select</a>
	1	0xx1		UNALLOCATED
	1	1xxx		<a href="#">Data-processing (3 source)</a>

## Data-processing (2 source)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	S	1	1	0	1	0	1	1	0	Rm				opcode				Rn				Rd								

sf	S	Decode fields opcode	Instruction Details	Architecture Version
		000001000000x	UNALLOCATED	-
		011xxx	UNALLOCATED	-
		1xxxxxx	UNALLOCATED	-
	0	0001010001xx	UNALLOCATED	-
	0	0011xx	UNALLOCATED	-
	1	00001x	UNALLOCATED	-
	10	0001xx000010	UDIVUNALLOCATED — 32-bit	-
	10	001xxx000011	SDIVUNALLOCATED — 32-bit	-
	10	01xxxx001000	LSLVUNALLOCATED — 32-bit	-
0		000000001001	LSRVUNALLOCATED — 32-bit	-
0	0	000010001010	UDIVASRV — 32-bit	-
0	0	000011001011	SDIVRORV — 32-bit	-
0	0	000100010x11	UNALLOCATED	-
0	0	00011x010000	CRC32B, CRC32H, CRC32W, CRC32XUNALLOCATED — CRC32B	-
0	0	001000010001	LSLVCRC32B, CRC32H, CRC32W, CRC32X — 32-bitCRC32H	-
0	0	001001010010	LSRVCRC32B, CRC32H, CRC32W, CRC32X — 32-bitCRC32W	-
0	0	00101001010100	ASRVCRC32CB, CRC32CH, CRC32CW, CRC32CX — 32-bitCRC32CB	-
0	0	00101101010101	RORVCRC32CB, CRC32CH, CRC32CW, CRC32CX — 32-bitCRC32CH	-
0	0	010x11010110	CRC32CB, CRC32CH, CRC32CW, CRC32CXUNALLOCATED — CRC32CW	-
0	0	010000	CRC32B, CRC32H, CRC32W, CRC32X — CRC32B	!
1	0	000010	UDIV — 64-bit	!
0	0	010001	CRC32B, CRC32H, CRC32W, CRC32X — CRC32H	!
1	0	000011	SDIV — 64-bit	!
0	0	010010	CRC32B, CRC32H, CRC32W, CRC32X — CRC32W	!
1	0	001000	LSLV — 64-bit	!
0	0	010100	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB	!
1	0	001001	LSRV — 64-bit	!
0	0	010101	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH	!
1	0	001010	ASRV — 64-bit	!
0	0	010110	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW	!
1	0	001011	RORV — 64-bit	!
1	0	000000001100	SUBPPACGA	ARMv8.5ARMv8.3
1	0	000010010xx0	UDIV — UNALLOCATED64-bit	-
1	0	000011010x0x	SDIV — UNALLOCATED64-bit	-
1	0	000100010011	— CRC32XIRGCRC32B, CRC32H, CRC32W, CRC32X	ARMv8.5-
1	0	000101010111	— CRC32CXGMICRC32CB, CRC32CH, CRC32CW, CRC32CX	ARMv8.5-
1	0	001000	LSLV — 64-bit	!
1	0	001001	LSRV — 64-bit	!
1	0	001010	ASRV — 64-bit	!
1	0	001011	RORV — 64-bit	!
1	0	001100	PACGA	ARMv8.3
1	0	010xx0	UNALLOCATED	!

Decode fields			Instruction Details			Architecture Version
sf	S	opcode				
1	0	010×0×	UNALLOCATED			-
1	0	010011	CRC32B, CRC32H, CRC32W, CRC32X — CRC32X			-
1	0	010111	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX			-
1	1	000000	SUBPS			ARMv8.5

## Data-processing (1 source)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	S	1	1	0	1	0	1	1	0	opcode2			opcode			Rn			Rd											

Decode fields			Instruction Details			Architecture Version
sf	S	opcode2	opcode	Rn		
			xx1xxx		UNALLOCATED	-
			x1xxxx		UNALLOCATED	-
			1xxxxx		UNALLOCATED	-
		xxxxx1			UNALLOCATED	-
		xxx1x			UNALLOCATED	-
		xx1xx			UNALLOCATED	-
		x1xxx			UNALLOCATED	-
		1xxxx			UNALLOCATED	-
	0	00000	00011x		UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00000	000000		RBIT — 32-bit	-
0	0	00000	000001		REV16 — 32-bit	-
0	0	00000	000010		REV — 32-bit	-
0	0	00000	000011		UNALLOCATED	-
0	0	00000	000100		CLZ — 32-bit	-
0	0	00000	000101		CLS — 32-bit	-
1	0	00000	000000		RBIT — 64-bit	-
1	0	00000	000001		REV16 — 64-bit	-
1	0	00000	000010		REV32	-
1	0	00000	000011		REV — 64-bit	-
1	0	00000	000100		CLZ — 64-bit	-
1	0	00000	000101		CLS — 64-bit	-
1	0	00001	000000		PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA	ARMv8.3
1	0	00001	000001		PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB	ARMv8.3
1	0	00001	000010		PACDA, PACDZA — PACDA	ARMv8.3
1	0	00001	000011		PACDB, PACDZB — PACDB	ARMv8.3
1	0	00001	000100		AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA	ARMv8.3
1	0	00001	000101		AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB	ARMv8.3
1	0	00001	000110		AUTDA, AUTDZA — AUTDA	ARMv8.3
1	0	00001	000111		AUTDB, AUTDZB — AUTDB	ARMv8.3
1	0	00001	001000	11111	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIZA	ARMv8.3
1	0	00001	001001	11111	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIZB	ARMv8.3
1	0	00001	001010	11111	PACDA, PACDZA — PACDZA	ARMv8.3

Decode fields					Instruction Details	Architecture Version
sf	S	opcode2	opcode	Rn		
1	0	00001	001011	11111	<a href="#">PACDB, PACDZB</a> — <a href="#">PACDZB</a>	ARMv8.3
1	0	00001	001100	11111	<a href="#">AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA</a> — <a href="#">AUTIZA</a>	ARMv8.3
1	0	00001	001101	11111	<a href="#">AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB</a> — <a href="#">AUTIZB</a>	ARMv8.3
1	0	00001	001110	11111	<a href="#">AUTDA, AUTDZA</a> — <a href="#">AUTDZA</a>	ARMv8.3
1	0	00001	001111	11111	<a href="#">AUTDB, AUTDZB</a> — <a href="#">AUTDZB</a>	ARMv8.3
1	0	00001	010000	11111	<a href="#">XPACD, XPACL, XPACLRI</a> — <a href="#">XPACI</a>	ARMv8.3
1	0	00001	010001	11111	<a href="#">XPACD, XPACL, XPACLRI</a> — <a href="#">XPACD</a>	ARMv8.3

### Logical (shifted register)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc	0	1	0	1	0		shift	N																						

Decode fields				Instruction Details
sf	opc	N	imm6	
0			1xxxxx	UNALLOCATED
0	00	0		<a href="#">AND (shifted register)</a> — <a href="#">32-bit</a>
0	00	1		<a href="#">BIC (shifted register)</a> — <a href="#">32-bit</a>
0	01	0		<a href="#">ORR (shifted register)</a> — <a href="#">32-bit</a>
0	01	1		<a href="#">ORN (shifted register)</a> — <a href="#">32-bit</a>
0	10	0		<a href="#">EOR (shifted register)</a> — <a href="#">32-bit</a>
0	10	1		<a href="#">EON (shifted register)</a> — <a href="#">32-bit</a>
0	11	0		<a href="#">ANDS (shifted register)</a> — <a href="#">32-bit</a>
0	11	1		<a href="#">BICS (shifted register)</a> — <a href="#">32-bit</a>
1	00	0		<a href="#">AND (shifted register)</a> — <a href="#">64-bit</a>
1	00	1		<a href="#">BIC (shifted register)</a> — <a href="#">64-bit</a>
1	01	0		<a href="#">ORR (shifted register)</a> — <a href="#">64-bit</a>
1	01	1		<a href="#">ORN (shifted register)</a> — <a href="#">64-bit</a>
1	10	0		<a href="#">EOR (shifted register)</a> — <a href="#">64-bit</a>
1	10	1		<a href="#">EON (shifted register)</a> — <a href="#">64-bit</a>
1	11	0		<a href="#">ANDS (shifted register)</a> — <a href="#">64-bit</a>
1	11	1		<a href="#">BICS (shifted register)</a> — <a href="#">64-bit</a>

### Add/subtract (shifted register)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	0	1	0	1	1	shift	0																						

Decode fields					Instruction Details
sf	op	S	shift	imm6	
			11		UNALLOCATED
0				1xxxxx	UNALLOCATED
0	0	0			<a href="#">ADD (shifted register)</a> — <a href="#">32-bit</a>
0	0	1			<a href="#">ADDS (shifted register)</a> — <a href="#">32-bit</a>
0	1	0			<a href="#">SUB (shifted register)</a> — <a href="#">32-bit</a>

Decode fields					Instruction Details
sf	op	S	shift	imm6	
0	1	1			<a href="#">SUBS (shifted register)</a> — 32-bit
1	0	0			<a href="#">ADD (shifted register)</a> — 64-bit
1	0	1			<a href="#">ADDS (shifted register)</a> — 64-bit
1	1	0			<a href="#">SUB (shifted register)</a> — 64-bit
1	1	1			<a href="#">SUBS (shifted register)</a> — 64-bit

### Add/subtract (extended register)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	0	1	0	1	1	opt	1	Rm						option			imm3			Rn				Rd					

Decode fields					Instruction Details
sf	op	S	opt	imm3	
				1x1	UNALLOCATED
				11x	UNALLOCATED
			x1		UNALLOCATED
			1x		UNALLOCATED
0	0	0	00		<a href="#">ADD (extended register)</a> — 32-bit
0	0	1	00		<a href="#">ADDS (extended register)</a> — 32-bit
0	1	0	00		<a href="#">SUB (extended register)</a> — 32-bit
0	1	1	00		<a href="#">SUBS (extended register)</a> — 32-bit
1	0	0	00		<a href="#">ADD (extended register)</a> — 64-bit
1	0	1	00		<a href="#">ADDS (extended register)</a> — 64-bit
1	1	0	00		<a href="#">SUB (extended register)</a> — 64-bit
1	1	1	00		<a href="#">SUBS (extended register)</a> — 64-bit

### Add/subtract (with carry)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				

Decode fields			Instruction Details
sf	op	S	
0	0	0	ADC — 32-bit
0	0	1	ADCS — 32-bit
0	1	0	SBC — 32-bit
0	1	1	SBCS — 32-bit
1	0	0	ADC — 64-bit
1	0	1	ADCS — 64-bit
1	1	0	SBC — 64-bit
1	1	1	SBCS — 64-bit

### Rotate right into flags

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	0	0	imm6						0	0	0	0	1	Rn				o2	mask				

Decode fields				Instruction Details	Architecture Version
sf	op	S	o2		
0				UNALLOCATED	-
1	0	0		UNALLOCATED	-
1	0	1	0	<a href="#">RMIF</a>	ARMv8.4
1	0	1	1	UNALLOCATED	-
1	1			UNALLOCATED	-

### Evaluate into flags

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	0	0	opcode2					sz	0	0	1	0	Rn			o3	mask						

Decode fields							Instruction Details	Architecture Version
sf	op	S	opcode2	sz	o3	mask		
0	0	0					UNALLOCATED	-
0	0	1	!= 000000				UNALLOCATED	-
0	0	1	000000		0	!= 1101	UNALLOCATED	-
0	0	1	000000		1		UNALLOCATED	-
0	0	1	000000	0	0	1101	<a href="#">SETF8, SETF16 — SETF8</a>	ARMv8.4
0	0	1	000000	1	0	1101	<a href="#">SETF8, SETF16 — SETF16</a>	ARMv8.4
0	1						UNALLOCATED	-
1							UNALLOCATED	-

### Conditional compare (register)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	1	0	Rm					cond			0	o2	Rn			o3	nzcw						

Decode fields					Instruction Details
sf	op	S	o2	o3	
				1	UNALLOCATED
			1		UNALLOCATED
		0			UNALLOCATED
0	0	1	0	0	CCMN (register) — 32-bit
0	1	1	0	0	CCMP (register) — 32-bit
1	0	1	0	0	CCMN (register) — 64-bit
1	1	1	0	0	CCMP (register) — 64-bit

### Conditional compare (immediate)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	1	0	imm5					cond			1	o2	Rn			o3	nzcw						

Decode fields					Instruction Details
sf	op	S	o2	o3	
				1	UNALLOCATED
			1		UNALLOCATED

Decode fields					Instruction Details
sf	op	S	o2	o3	
		0			UNALLOCATED
0	0	1	0	0	CCMN (immediate) — 32-bit
0	1	1	0	0	CCMP (immediate) — 32-bit
1	0	1	0	0	CCMN (immediate) — 64-bit
1	1	1	0	0	CCMP (immediate) — 64-bit

## Conditional select

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	1	0	0	Rm				cond				op2				Rn				Rd				

Decode fields				Instruction Details
sf	op	S	op2	
			1x	UNALLOCATED
		1		UNALLOCATED
0	0	0	00	CSEL — 32-bit
0	0	0	01	CSINC — 32-bit
0	1	0	00	CSINV — 32-bit
0	1	0	01	CSNEG — 32-bit
1	0	0	00	CSEL — 64-bit
1	0	0	01	CSINC — 64-bit
1	1	0	00	CSINV — 64-bit
1	1	0	01	CSNEG — 64-bit

## Data-processing (3 source)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		op54		1	1	0	1	1	op31				Rm				o0		Ra				Rn				Rd				

Decode fields				Instruction Details
sf	op54	op31	o0	
	00	010	1	UNALLOCATED
	00	011		UNALLOCATED
	00	100		UNALLOCATED
	00	110	1	UNALLOCATED
	00	111		UNALLOCATED
	01			UNALLOCATED
	1x			UNALLOCATED
0	00	000	0	MADD — 32-bit
0	00	000	1	MSUB — 32-bit
0	00	001	0	UNALLOCATED
0	00	001	1	UNALLOCATED
0	00	010	0	UNALLOCATED
0	00	101	0	UNALLOCATED
0	00	101	1	UNALLOCATED
0	00	110	0	UNALLOCATED

sf	Decode fields			Instruction Details
	op54	op31	o0	
1	00	000	0	MADD — 64-bit
1	00	000	1	MSUB — 64-bit
1	00	001	0	SMADDL
1	00	001	1	SMSUBL
1	00	010	0	SMULH
1	00	101	0	UMADDL
1	00	101	1	UMSUBL
1	00	110	0	UMULH

## Data Processing -- Scalar Floating-Point and Advanced SIMD

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0				111			op1		op2			op3																			

Decode fields				Instruction details	Architecture version
op0	op1	op2	op3		
0000	0x	x101	00xxxxx10	UNALLOCATED	-
0010	0x	x101	00xxxxx10	UNALLOCATED	-
0100	0x	x101	00xxxxx10	<a href="#">Cryptographic AES</a>	-
0101	0x	x0xx	xxx0xxx00	<a href="#">Cryptographic three-register SHA</a>	-
0101	0x	x0xx	xxx0xxx10	UNALLOCATED	-
0101	0x	x101	00xxxxx10	<a href="#">Cryptographic two-register SHA</a>	-
0110	0x	x101	00xxxxx10	UNALLOCATED	-
0111	0x	x0xx	xxx0xxx00	UNALLOCATED	-
0111	0x	x101	00xxxxx10	UNALLOCATED	-
01x1	00	00xx	xxx0xxx01	<a href="#">Advanced SIMD scalar copy</a>	-
01x1	01	00xx	xxx0xxx01	UNALLOCATED	-
01x1	0x	0111	00xxxxx10	UNALLOCATED	-
01x1	0x	10xx	xxx00xxx1	<a href="#">Advanced SIMD scalar three same FP16</a>	ARMv8.2
01x1	0x	10xx	xxx01xxx1	UNALLOCATED	-
01x1	0x	1111	00xxxxx10	<a href="#">Advanced SIMD scalar two-register miscellaneous FP16</a>	ARMv8.2
01x1	0x	x0xx	xxx1xxx00	UNALLOCATED	-
01x1	0x	x0xx	xxx1xxx01	<a href="#">Advanced SIMD scalar three same extra</a>	ARMv8.1
01x1	0x	x100	00xxxxx10	<a href="#">Advanced SIMD scalar two-register miscellaneous</a>	-
01x1	0x	x110	00xxxxx10	<a href="#">Advanced SIMD scalar pairwise</a>	ARMv8.2
01x1	0x	x1xx	1xxxxxx10	UNALLOCATED	-
01x1	0x	x1xx	x1xxxxx10	UNALLOCATED	-
01x1	0x	x1xx	xxxxxxx00	<a href="#">Advanced SIMD scalar three different</a>	-
01x1	0x	x1xx	xxxxxxx01	<a href="#">Advanced SIMD scalar three same</a>	-
01x1	10		xxxxxxx01	<a href="#">Advanced SIMD scalar shift by immediate</a>	-
01x1	11		xxxxxxx01	UNALLOCATED	-
01x1	1x		xxxxxxx00	<a href="#">Advanced SIMD scalar x indexed element</a>	ARMv8.2
0x00	0x	x0xx	xxx0xxx00	<a href="#">Advanced SIMD table lookup</a>	-
0x00	0x	x0xx	xxx0xxx10	<a href="#">Advanced SIMD permute</a>	-
0x10	0x	x0xx	xxx0xxx00	<a href="#">Advanced SIMD extract</a>	-
0xx0	00	00xx	xxx0xxx01	<a href="#">Advanced SIMD copy</a>	-



0xx0	01	00xx	xxx0xxxx1	UNALLOCATED	-
0xx0	0x	0111	00xxxxx10	UNALLOCATED	-
0xx0	0x	10xx	xxx00xxx1	<a href="#">Advanced SIMD three same (FP16)</a>	ARMv8.2
0xx0	0x	10xx	xxx01xxx1	UNALLOCATED	-
0xx0	0x	1111	00xxxxx10	<a href="#">Advanced SIMD two-register miscellaneous (FP16)</a>	ARMv8.2
0xx0	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0xx0	0x	x0xx	xxx1xxxx1	<a href="#">Advanced SIMD three same extra</a>	ARMv8.2
0xx0	0x	x100	00xxxxx10	<a href="#">Advanced SIMD two-register miscellaneous</a>	ARMv8.5
0xx0	0x	x110	00xxxxx10	<a href="#">Advanced SIMD across lanes</a>	ARMv8.2
0xx0	0x	x1xx	1xxxxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	x1xxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	xxxxxxx00	<a href="#">Advanced SIMD three different</a>	-
0xx0	0x	x1xx	xxxxxxx1	<a href="#">Advanced SIMD three same</a>	ARMv8.2
0xx0	10	0000	xxxxxxx1	<a href="#">Advanced SIMD modified immediate</a>	ARMv8.2
0xx0	10	!= 0000	xxxxxxx1	<a href="#">Advanced SIMD shift by immediate</a>	-
0xx0	11		xxxxxxx1	UNALLOCATED	-
0xx0	1x		xxxxxxx0	<a href="#">Advanced SIMD vector x indexed element</a>	ARMv8.2
1100	00	10xx	xxx10xxxx	<a href="#">Cryptographic three-register, imm2</a>	ARMv8.2
1100	00	11xx	xxx1x00xx	<a href="#">Cryptographic three-register SHA 512</a>	ARMv8.2
1100	00		xxx0xxxxx	<a href="#">Cryptographic four-register</a>	ARMv8.2
1100	01	00xx		<a href="#">XAR</a>	ARMv8.2
1100	01	1000	0001000xx	<a href="#">Cryptographic two-register SHA 512</a>	ARMv8.2
11x1				UNALLOCATED	-
1xx0	1x			UNALLOCATED	-
x0x1	0x	x0xx		<a href="#">Conversion between floating-point and fixed-point</a>	ARMv8.2
x0x1	0x	x1xx	xxx000000	<a href="#">Conversion between floating-point and integer</a>	ARMv8.3
x0x1	0x	x1xx	xxx100000	UNALLOCATED	-
x0x1	0x	x1xx	xxx10000	<a href="#">Floating-point data-processing (1 source)</a>	ARMv8.5ARMv8.2
x0x1	0x	x1xx	xxxxx1000	<a href="#">Floating-point compare</a>	ARMv8.2
x0x1	0x	x1xx	xxxxxx100	<a href="#">Floating-point immediate</a>	ARMv8.2
x0x1	0x	x1xx	xxxxxxx01	<a href="#">Floating-point conditional compare</a>	ARMv8.2
x0x1	0x	x1xx	xxxxxxx10	<a href="#">Floating-point data-processing (2 source)</a>	ARMv8.2
x0x1	0x	x1xx	xxxxxxx11	<a href="#">Floating-point conditional select</a>	ARMv8.2
x0x1	1x			<a href="#">Floating-point data-processing (3 source)</a>	ARMv8.2

## Cryptographic AES

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	size		1	0	1	0	0	opcode				1		0	Rn				Rd					

Decode fields size	opcode	Instruction Details
	x1xxx	UNALLOCATED
	000xx	UNALLOCATED
	1xxxx	UNALLOCATED
x1		UNALLOCATED
00	00100	<a href="#">AESE</a>
00	00101	<a href="#">AESD</a>

Decode fields size	opcode	Instruction Details
00	00110	<a href="#">AESMC</a>
00	00111	<a href="#">AESIMC</a>
1x		UNALLOCATED

### Cryptographic three-register SHA

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	0					Rm		0		opcode	0	0					Rn				Rd		

Decode fields size	opcode	Instruction Details
	111	UNALLOCATED
x1		UNALLOCATED
00	000	<a href="#">SHA1C</a>
00	001	<a href="#">SHA1P</a>
00	010	<a href="#">SHA1M</a>
00	011	<a href="#">SHA1SU0</a>
00	100	<a href="#">SHA256H</a>
00	101	<a href="#">SHA256H2</a>
00	110	<a href="#">SHA256SU1</a>
1x		UNALLOCATED

### Cryptographic two-register SHA

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	0	1	0	0					opcode			1	0				Rn				Rd	

Decode fields size	opcode	Instruction Details
	xx1xx	UNALLOCATED
	x1xxx	UNALLOCATED
	1xxxx	UNALLOCATED
x1		UNALLOCATED
00	00000	<a href="#">SHA1H</a>
00	00001	<a href="#">SHA1SU1</a>
00	00010	<a href="#">SHA256SU0</a>
00	00011	UNALLOCATED
1x		UNALLOCATED

### Advanced SIMD scalar copy

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	op	1	1	1	1	0	0	0	0					imm5		0					1				Rn				Rd	

Decode fields			Instruction Details
op	imm5	imm4	
0		xxx1	UNALLOCATED
0		xx1x	UNALLOCATED
0		x1xx	UNALLOCATED
0		0000	<a href="#">DUP (element)</a>
0		1xxx	UNALLOCATED
0	x0000	0000	UNALLOCATED
1			UNALLOCATED

### Advanced SIMD scalar three same FP16

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	a	1	0	Rm				0	0	opcode				1	Rn				Rd					

Decode fields			Instruction Details	Architecture Version
U	a	opcode		
		110	UNALLOCATED	-
	1	011	UNALLOCATED	-
0	0	011	<a href="#">FMULX</a>	ARMv8.2
0	0	100	<a href="#">FCMEQ (register)</a>	ARMv8.2
0	0	101	UNALLOCATED	-
0	0	111	<a href="#">FRECPS</a>	ARMv8.2
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	111	<a href="#">FRSQRTS</a>	ARMv8.2
1	0	011	UNALLOCATED	-
1	0	100	<a href="#">FCMGE (register)</a>	ARMv8.2
1	0	101	<a href="#">FACGE</a>	ARMv8.2
1	0	111	UNALLOCATED	-
1	1	010	<a href="#">FABD</a>	ARMv8.2
1	1	100	<a href="#">FCMGT (register)</a>	ARMv8.2
1	1	101	<a href="#">FACGT</a>	ARMv8.2
1	1	111	UNALLOCATED	-

### Advanced SIMD scalar two-register miscellaneous FP16

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	a	1	1	1	1	0	0	opcode				1	0	Rn				Rd						

Decode fields			Instruction Details	Architecture Version
U	a	opcode		
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-

Decode fields			Instruction Details	Architecture Version
U	a	opcode		
	0	11111	UNALLOCATED	-
	1	01111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11010	<a href="#">FCVTNS (vector)</a>	ARMv8.2
0	0	11011	<a href="#">FCVTMS (vector)</a>	ARMv8.2
0	0	11100	<a href="#">FCVTAS (vector)</a>	ARMv8.2
0	0	11101	<a href="#">SCVTF (vector, integer)</a>	ARMv8.2
0	1	01100	<a href="#">FCMGT (zero)</a>	ARMv8.2
0	1	01101	<a href="#">FCMEQ (zero)</a>	ARMv8.2
0	1	01110	<a href="#">FCMLT (zero)</a>	ARMv8.2
0	1	11010	<a href="#">FCVTPS (vector)</a>	ARMv8.2
0	1	11011	<a href="#">FCVTZS (vector, integer)</a>	ARMv8.2
0	1	11101	<a href="#">FRECPE</a>	ARMv8.2
0	1	11111	<a href="#">FRECPX</a>	ARMv8.2
1	0	11010	<a href="#">FCVTNU (vector)</a>	ARMv8.2
1	0	11011	<a href="#">FCVTMU (vector)</a>	ARMv8.2
1	0	11100	<a href="#">FCVTAU (vector)</a>	ARMv8.2
1	0	11101	<a href="#">UCVTF (vector, integer)</a>	ARMv8.2
1	1	01100	<a href="#">FCMGE (zero)</a>	ARMv8.2
1	1	01101	<a href="#">FCMLE (zero)</a>	ARMv8.2
1	1	01110	UNALLOCATED	-
1	1	11010	<a href="#">FCVTPU (vector)</a>	ARMv8.2
1	1	11011	<a href="#">FCVTZU (vector, integer)</a>	ARMv8.2
1	1	11101	<a href="#">FRSQRT</a>	ARMv8.2
1	1	11111	UNALLOCATED	-

### Advanced SIMD scalar three same extra

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size	0		Rm		1		opcode	1		Rn													Rd

Decode fields			Instruction Details	Architecture Version
U	a	opcode		
		001x	UNALLOCATED	-
		01xx	UNALLOCATED	-
		1xxx	UNALLOCATED	-
0		0000	UNALLOCATED	-
0		0001	UNALLOCATED	-
1		0000	<a href="#">SQRDMLAH (vector)</a>	ARMv8.1
1		0001	<a href="#">SQRDMLSH (vector)</a>	ARMv8.1

### Advanced SIMD scalar two-register miscellaneous

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size	1	0	0	0	0		opcode		1	0													Rd

Decode fields		opcode	Instruction Details
U	size		
		0000x	UNALLOCATED
		00010	UNALLOCATED
		0010x	UNALLOCATED
		00110	UNALLOCATED
		01111	UNALLOCATED
		1000x	UNALLOCATED
		10011	UNALLOCATED
		10101	UNALLOCATED
		10111	UNALLOCATED
		1100x	UNALLOCATED
		11110	UNALLOCATED
	0x	011xx	UNALLOCATED
	0x	11111	UNALLOCATED
	1x	10110	UNALLOCATED
	1x	11100	UNALLOCATED
0		00011	<a href="#">SUQADD</a>
0		00111	<a href="#">SQABS</a>
0		01000	<a href="#">CMGT (zero)</a>
0		01001	<a href="#">CMEQ (zero)</a>
0		01010	<a href="#">CMLT (zero)</a>
0		01011	<a href="#">ABS</a>
0		10010	UNALLOCATED
0		10100	<a href="#">SQXTN, SQXTN2</a>
0	0x	10110	UNALLOCATED
0	0x	11010	<a href="#">FCVTNS (vector)</a>
0	0x	11011	<a href="#">FCVTMS (vector)</a>
0	0x	11100	<a href="#">FCVTAS (vector)</a>
0	0x	11101	<a href="#">SCVTF (vector, integer)</a>
0	1x	01100	<a href="#">FCMGT (zero)</a>
0	1x	01101	<a href="#">FCMEQ (zero)</a>
0	1x	01110	<a href="#">FCMLT (zero)</a>
0	1x	11010	<a href="#">FCVTPS (vector)</a>
0	1x	11011	<a href="#">FCVTZS (vector, integer)</a>
0	1x	11101	<a href="#">FRECPE</a>
0	1x	11111	<a href="#">FRECPX</a>
1		00011	<a href="#">USQADD</a>
1		00111	<a href="#">SQNEG</a>
1		01000	<a href="#">CMGE (zero)</a>
1		01001	<a href="#">CMLE (zero)</a>
1		01010	UNALLOCATED
1		01011	<a href="#">NEG (vector)</a>
1		10010	<a href="#">SQXTUN, SQXTUN2</a>
1		10100	<a href="#">UQXTN, UQXTN2</a>
1	0x	10110	<a href="#">FCVTXN, FCVTXN2</a>
1	0x	11010	<a href="#">FCVTNU (vector)</a>
1	0x	11011	<a href="#">FCVTMU (vector)</a>
1	0x	11100	<a href="#">FCVTAU (vector)</a>

Decode fields			Instruction Details
U	size	opcode	
1	0x	11101	<a href="#">UCVTF (vector, integer)</a>
1	1x	01100	<a href="#">FCMGE (zero)</a>
1	1x	01101	<a href="#">FCMLE (zero)</a>
1	1x	01110	UNALLOCATED
1	1x	11010	<a href="#">FCVTPU (vector)</a>
1	1x	11011	<a href="#">FCVTZU (vector, integer)</a>
1	1x	11101	<a href="#">ERSQRTE</a>
1	1x	11111	UNALLOCATED

### Advanced SIMD scalar pairwise

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size	1	1	0	0	0	opcode			1	0	Rn				Rd								

Decode fields			Instruction Details	Architecture Version
U	size	opcode		
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11010	UNALLOCATED	-
		111xx	UNALLOCATED	-
	1x	01101	UNALLOCATED	-
0		11011	<a href="#">ADDP (scalar)</a>	-
0	00	01100	<a href="#">FMAXNMP (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	00	01101	<a href="#">FADDP (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	00	01111	<a href="#">FMAXP (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	01	01100	UNALLOCATED	-
0	01	01101	UNALLOCATED	-
0	01	01111	UNALLOCATED	-
0	10	01100	<a href="#">FMINNMP (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	10	01111	<a href="#">FMINP (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	11	01100	UNALLOCATED	-
0	11	01111	UNALLOCATED	-
1		11011	UNALLOCATED	-
1	0x	01100	<a href="#">FMAXNMP (scalar)</a> — <a href="#">single-precision and double-precision</a>	-
1	0x	01101	<a href="#">FADDP (scalar)</a> — <a href="#">single-precision and double-precision</a>	-
1	0x	01111	<a href="#">FMAXP (scalar)</a> — <a href="#">single-precision and double-precision</a>	-
1	1x	01100	<a href="#">FMINNMP (scalar)</a> — <a href="#">single-precision and double-precision</a>	-
1	1x	01111	<a href="#">FMINP (scalar)</a> — <a href="#">single-precision and double-precision</a>	-

### Advanced SIMD scalar three different

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size	1	Rm			opcode			0	0	Rn				Rd									

Decode fields		Instruction Details
U	opcode	
	00xx	UNALLOCATED
	01xx	UNALLOCATED
	1000	UNALLOCATED
	1010	UNALLOCATED
	1100	UNALLOCATED
	111x	UNALLOCATED
0	1001	<a href="#">SQDMLAL, SQDMLAL2 (vector)</a>
0	1011	<a href="#">SQDMLSL, SQDMLSL2 (vector)</a>
0	1101	<a href="#">SQDMULL, SQDMULL2 (vector)</a>
1	1001	UNALLOCATED
1	1011	UNALLOCATED
1	1101	UNALLOCATED

### Advanced SIMD scalar three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	1	U	1	1	1	1	0	size	1	Rm						opcode						1	Rn						Rd					

Decode fields		Instruction Details
U	size opcode	
	00000	UNALLOCATED
	0001x	UNALLOCATED
	00100	UNALLOCATED
	011xx	UNALLOCATED
	1001x	UNALLOCATED
	1x 11011	UNALLOCATED
0	00001	<a href="#">SQADD</a>
0	00101	<a href="#">SQSUB</a>
0	00110	<a href="#">CMGT (register)</a>
0	00111	<a href="#">CMGE (register)</a>
0	01000	<a href="#">SSHLL</a>
0	01001	<a href="#">SQSHLL (register)</a>
0	01010	<a href="#">SRSHLL</a>
0	01011	<a href="#">SQRSHL</a>
0	10000	<a href="#">ADD (vector)</a>
0	10001	<a href="#">CMTST</a>
0	10100	UNALLOCATED
0	10101	UNALLOCATED
0	10110	<a href="#">SQDMULH (vector)</a>
0	10111	UNALLOCATED
0	0x 11000	UNALLOCATED
0	0x 11001	UNALLOCATED
0	0x 11010	UNALLOCATED
0	0x 11011	<a href="#">FMULX</a>
0	0x 11100	<a href="#">FCMEQ (register)</a>
0	0x 11101	UNALLOCATED
0	0x 11110	UNALLOCATED

U	Decode fields		Instruction Details
	size	opcode	
0	0x	11111	<a href="#">FRECPS</a>
0	1x	11000	UNALLOCATED
0	1x	11001	UNALLOCATED
0	1x	11010	UNALLOCATED
0	1x	11100	UNALLOCATED
0	1x	11101	UNALLOCATED
0	1x	11110	UNALLOCATED
0	1x	11111	<a href="#">FRSQRTS</a>
1		00001	<a href="#">UQADD</a>
1		00101	<a href="#">UQSUB</a>
1		00110	<a href="#">CMHI (register)</a>
1		00111	<a href="#">CMHS (register)</a>
1		01000	<a href="#">USHL</a>
1		01001	<a href="#">UQSHL (register)</a>
1		01010	<a href="#">URSHL</a>
1		01011	<a href="#">UQRSHL</a>
1		10000	<a href="#">SUB (vector)</a>
1		10001	<a href="#">CMEQ (register)</a>
1		10100	UNALLOCATED
1		10101	UNALLOCATED
1		10110	<a href="#">SQRDMULH (vector)</a>
1		10111	UNALLOCATED
1	0x	11000	UNALLOCATED
1	0x	11001	UNALLOCATED
1	0x	11010	UNALLOCATED
1	0x	11011	UNALLOCATED
1	0x	11100	<a href="#">FCMGE (register)</a>
1	0x	11101	<a href="#">FACGE</a>
1	0x	11110	UNALLOCATED
1	0x	11111	UNALLOCATED
1	1x	11000	UNALLOCATED
1	1x	11001	UNALLOCATED
1	1x	11010	<a href="#">FABD</a>
1	1x	11100	<a href="#">FCMGT (register)</a>
1	1x	11101	<a href="#">FACGT</a>
1	1x	11110	UNALLOCATED
1	1x	11111	UNALLOCATED

### Advanced SIMD scalar shift by immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	1	0	immh			immb			opcode						1	Rn				Rd					

U	Decode fields		Instruction Details
	immh	opcode	
	!= 0000	00001	UNALLOCATED
	!= 0000	00011	UNALLOCATED



U	Decode fields		Instruction Details
	immh	opcode	
	!= 0000	00101	UNALLOCATED
	!= 0000	00111	UNALLOCATED
	!= 0000	01001	UNALLOCATED
	!= 0000	01011	UNALLOCATED
	!= 0000	01101	UNALLOCATED
	!= 0000	01111	UNALLOCATED
	!= 0000	101xx	UNALLOCATED
	!= 0000	110xx	UNALLOCATED
	!= 0000	11101	UNALLOCATED
	!= 0000	11110	UNALLOCATED
	0000		UNALLOCATED
0	!= 0000	00000	<a href="#">SSHR</a>
0	!= 0000	00010	<a href="#">SSRA</a>
0	!= 0000	00100	<a href="#">SRSHR</a>
0	!= 0000	00110	<a href="#">SRSRA</a>
0	!= 0000	01000	UNALLOCATED
0	!= 0000	01010	<a href="#">SHL</a>
0	!= 0000	01100	UNALLOCATED
0	!= 0000	01110	<a href="#">SQSHL (immediate)</a>
0	!= 0000	10000	UNALLOCATED
0	!= 0000	10001	UNALLOCATED
0	!= 0000	10010	<a href="#">SQSHRN, SQSHRN2</a>
0	!= 0000	10011	<a href="#">SQRSHRN, QQRSHRN2</a>
0	!= 0000	11100	<a href="#">SCVTF (vector, fixed-point)</a>
0	!= 0000	11111	<a href="#">FCVTZS (vector, fixed-point)</a>
1	!= 0000	00000	<a href="#">USHR</a>
1	!= 0000	00010	<a href="#">USRA</a>
1	!= 0000	00100	<a href="#">URSHR</a>
1	!= 0000	00110	<a href="#">URSRA</a>
1	!= 0000	01000	<a href="#">SRI</a>
1	!= 0000	01010	<a href="#">SLI</a>
1	!= 0000	01100	<a href="#">SQSHLU</a>
1	!= 0000	01110	<a href="#">UQSHL (immediate)</a>
1	!= 0000	10000	<a href="#">SQSHRUN, SQSHRUN2</a>
1	!= 0000	10001	<a href="#">SQRSHRUN, QQRSHRUN2</a>
1	!= 0000	10010	<a href="#">UQSHRN, UQSHRN2</a>
1	!= 0000	10011	<a href="#">UQRSHRN, UQRSHRN2</a>
1	!= 0000	11100	<a href="#">UCVTF (vector, fixed-point)</a>
1	!= 0000	11111	<a href="#">FCVTZU (vector, fixed-point)</a>

### Advanced SIMD scalar x indexed element

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	1	size	L	M	Rm			opcode			H	0	Rn				Rd								

Decode fields			Instruction Details	Architecture Version
U	size	opcode		
		0000	UNALLOCATED	-
		0010	UNALLOCATED	-
		0100	UNALLOCATED	-
		0110	UNALLOCATED	-
		1000	UNALLOCATED	-
		1010	UNALLOCATED	-
		1110	UNALLOCATED	-
	01	0001	UNALLOCATED	-
	01	0101	UNALLOCATED	-
	01	1001	UNALLOCATED	-
0		0011	<a href="#">SQDMLAL, SQDMLAL2 (by element)</a>	-
0		0111	<a href="#">SQDMLSL, SQDMLSL2 (by element)</a>	-
0		1011	<a href="#">SQDMULL, SQDMULL2 (by element)</a>	-
0		1100	<a href="#">SQDMULH (by element)</a>	-
0		1101	<a href="#">SQRDMULH (by element)</a>	-
0		1111	UNALLOCATED	-
0	00	0001	<a href="#">FMLA (by element)</a> — <a href="#">half-precision</a>	ARMv8.2
0	00	0101	<a href="#">FMLS (by element)</a> — <a href="#">half-precision</a>	ARMv8.2
0	00	1001	<a href="#">FMUL (by element)</a> — <a href="#">half-precision</a>	ARMv8.2
0	1x	0001	<a href="#">FMLA (by element)</a> — <a href="#">single-precision and double-precision</a>	-
0	1x	0101	<a href="#">FMLS (by element)</a> — <a href="#">single-precision and double-precision</a>	-
0	1x	1001	<a href="#">FMUL (by element)</a> — <a href="#">single-precision and double-precision</a>	-
1		0011	UNALLOCATED	-
1		0111	UNALLOCATED	-
1		1011	UNALLOCATED	-
1		1100	UNALLOCATED	-
1		1101	<a href="#">SQRDMLAH (by element)</a>	ARMv8.1
1		1111	<a href="#">SQRDMLSH (by element)</a>	ARMv8.1
1	00	0001	UNALLOCATED	-
1	00	0101	UNALLOCATED	-
1	00	1001	<a href="#">FMULX (by element)</a> — <a href="#">half-precision</a>	ARMv8.2
1	1x	0001	UNALLOCATED	-
1	1x	0101	UNALLOCATED	-
1	1x	1001	<a href="#">FMULX (by element)</a> — <a href="#">single-precision and double-precision</a>	-

### Advanced SIMD table lookup

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	op2	0		Rm		0	len	op	0	0		Rn												

Decode fields			Instruction Details
op2	len	op	
x1			UNALLOCATED
00	00	0	<a href="#">TBL</a> — <a href="#">single register table</a>
00	00	1	<a href="#">TBX</a> — <a href="#">single register table</a>
00	01	0	<a href="#">TBL</a> — <a href="#">two register table</a>
00	01	1	<a href="#">TBX</a> — <a href="#">two register table</a>

Decode fields			Instruction Details
op2	len	op	
00	10	0	<a href="#">TBL</a> — <a href="#">three register table</a>
00	10	1	<a href="#">TBX</a> — <a href="#">three register table</a>
00	11	0	<a href="#">TBL</a> — <a href="#">four register table</a>
00	11	1	<a href="#">TBX</a> — <a href="#">four register table</a>
1x			UNALLOCATED

## Advanced SIMD permute

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	0				Rm		0		opcode	1	0												Rd

Decode fields opcode	Instruction Details
000	UNALLOCATED
001	<a href="#">UZP1</a>
010	<a href="#">TRN1</a>
011	<a href="#">ZIP1</a>
100	UNALLOCATED
101	<a href="#">UZP2</a>
110	<a href="#">TRN2</a>
111	<a href="#">ZIP2</a>

## Advanced SIMD extract

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	op2	0				Rm		0		imm4		0												Rd

Decode fields op2	Instruction Details
x1	UNALLOCATED
00	<a href="#">EXT</a>
1x	UNALLOCATED

## Advanced SIMD copy

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	0	0	0	0					imm5		0		imm4		1										Rd

Decode fields				Instruction Details
Q	op	imm5	imm4	
		x0000		UNALLOCATED
	0		0000	<a href="#">DUP (element)</a>
	0		0001	<a href="#">DUP (general)</a>
	0		0010	UNALLOCATED
	0		0100	UNALLOCATED
	0		0110	UNALLOCATED

Decode fields				Instruction Details
Q	op	imm5	imm4	
	0		1xxx	UNALLOCATED
0	0		0011	UNALLOCATED
0	0		0101	<a href="#">SMOV</a>
0	0		0111	<a href="#">UMOV</a>
0	1			UNALLOCATED
1	0		0011	<a href="#">INS (general)</a>
1	0		0101	<a href="#">SMOV</a>
1	0	x1000	0111	<a href="#">UMOV</a>
1	1			<a href="#">INS (element)</a>

### Advanced SIMD three same (FP16)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	a	1	0	Rm				0	0	opcode			1	Rn				Rd						

Decode fields			Instruction Details	Architecture Version
U	a	opcode		
0	0	000	<a href="#">FMAXNM (vector)</a>	ARMv8.2
0	0	001	<a href="#">FMLA (vector)</a>	ARMv8.2
0	0	010	<a href="#">FADD (vector)</a>	ARMv8.2
0	0	011	<a href="#">FMULX</a>	ARMv8.2
0	0	100	<a href="#">FCMEQ (register)</a>	ARMv8.2
0	0	101	UNALLOCATED	-
0	0	110	<a href="#">FMAX (vector)</a>	ARMv8.2
0	0	111	<a href="#">FRECPS</a>	ARMv8.2
0	1	000	<a href="#">FMINNM (vector)</a>	ARMv8.2
0	1	001	<a href="#">FMLS (vector)</a>	ARMv8.2
0	1	010	<a href="#">FSUB (vector)</a>	ARMv8.2
0	1	011	UNALLOCATED	-
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	110	<a href="#">FMIN (vector)</a>	ARMv8.2
0	1	111	<a href="#">FRSQRTS</a>	ARMv8.2
1	0	000	<a href="#">FMAXNMP (vector)</a>	ARMv8.2
1	0	001	UNALLOCATED	-
1	0	010	<a href="#">FADDP (vector)</a>	ARMv8.2
1	0	011	<a href="#">FMUL (vector)</a>	ARMv8.2
1	0	100	<a href="#">FCMGE (register)</a>	ARMv8.2
1	0	101	<a href="#">FACGE</a>	ARMv8.2
1	0	110	<a href="#">FMAXP (vector)</a>	ARMv8.2
1	0	111	<a href="#">FDIV (vector)</a>	ARMv8.2
1	1	000	<a href="#">FMINNMP (vector)</a>	ARMv8.2
1	1	001	UNALLOCATED	-
1	1	010	<a href="#">FABD</a>	ARMv8.2
1	1	011	UNALLOCATED	-
1	1	100	<a href="#">FCMGT (register)</a>	ARMv8.2
1	1	101	<a href="#">FACGT</a>	ARMv8.2

Decode fields			Instruction Details	Architecture Version
U	a	opcode		
1	1	110	<a href="#">FMINP (vector)</a>	ARMv8.2
1	1	111	UNALLOCATED	-

### Advanced SIMD two-register miscellaneous (FP16)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	a	1	1	1	1	0	0	opcode				1 0		Rn				Rd						

Decode fields			Instruction Details	Architecture Version
U	a	opcode		
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11000	<a href="#">FRINTN (vector)</a>	ARMv8.2
0	0	11001	<a href="#">FRINTM (vector)</a>	ARMv8.2
0	0	11010	<a href="#">FCVTNS (vector)</a>	ARMv8.2
0	0	11011	<a href="#">FCVTMS (vector)</a>	ARMv8.2
0	0	11100	<a href="#">FCVTAS (vector)</a>	ARMv8.2
0	0	11101	<a href="#">SCVTF (vector, integer)</a>	ARMv8.2
0	1	01100	<a href="#">FCMGT (zero)</a>	ARMv8.2
0	1	01101	<a href="#">FCMEQ (zero)</a>	ARMv8.2
0	1	01110	<a href="#">FCMLT (zero)</a>	ARMv8.2
0	1	01111	<a href="#">FABS (vector)</a>	ARMv8.2
0	1	11000	<a href="#">FRINTP (vector)</a>	ARMv8.2
0	1	11001	<a href="#">FRINTZ (vector)</a>	ARMv8.2
0	1	11010	<a href="#">FCVTPS (vector)</a>	ARMv8.2
0	1	11011	<a href="#">FCVTZS (vector, integer)</a>	ARMv8.2
0	1	11101	<a href="#">FRECPE</a>	ARMv8.2
0	1	11111	UNALLOCATED	-
1	0	11000	<a href="#">FRINTA (vector)</a>	ARMv8.2
1	0	11001	<a href="#">FRINTX (vector)</a>	ARMv8.2
1	0	11010	<a href="#">FCVTNU (vector)</a>	ARMv8.2
1	0	11011	<a href="#">FCVTMU (vector)</a>	ARMv8.2
1	0	11100	<a href="#">FCVTAU (vector)</a>	ARMv8.2
1	0	11101	<a href="#">UCVTF (vector, integer)</a>	ARMv8.2
1	1	01100	<a href="#">FCMGE (zero)</a>	ARMv8.2
1	1	01101	<a href="#">FCMLE (zero)</a>	ARMv8.2
1	1	01110	UNALLOCATED	-
1	1	01111	<a href="#">FNEG (vector)</a>	ARMv8.2
1	1	11000	UNALLOCATED	-
1	1	11001	<a href="#">FRINTI (vector)</a>	ARMv8.2
1	1	11010	<a href="#">FCVTPI (vector)</a>	ARMv8.2
1	1	11011	<a href="#">FCVTZI (vector, integer)</a>	ARMv8.2

Decode fields			Instruction Details	Architecture Version
U	a	opcode		
1	1	11101	<a href="#">FRSQORTE</a>	ARMv8.2
1	1	11111	<a href="#">FSQRT (vector)</a>	ARMv8.2

### Advanced SIMD three same extra

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size	0				Rm			1			opcode	1											Rd

Decode fields			Instruction Details	Architecture Version
U		opcode		
		0011	UNALLOCATED	-
		01xx	UNALLOCATED	-
0		0000	UNALLOCATED	-
0		0001	UNALLOCATED	-
0		0010	<a href="#">SDOT (vector)</a>	ARMv8.2
0		1xxx	UNALLOCATED	-
1		0000	<a href="#">SQRDMLAH (vector)</a>	ARMv8.1
1		0001	<a href="#">SQRDMLSH (vector)</a>	ARMv8.1
1		0010	<a href="#">UDOT (vector)</a>	ARMv8.2
1		10xx	<a href="#">FCMLA</a>	ARMv8.3
1		11x0	<a href="#">FCADD</a>	ARMv8.3
1		11x1	UNALLOCATED	-

### Advanced SIMD two-register miscellaneous

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size	1	0	0	0	0						opcode	1	0										Rd

Decode fields			Instruction Details	Architecture Version
U	size	opcode		
		1000x	UNALLOCATED	-
		10101	UNALLOCATED	-
	0x	011xx	UNALLOCATED	-
	1x	10110	UNALLOCATED	-
	1x	10111	UNALLOCATED	-
	1x	11110	UNALLOCATED	-
0		00000	<a href="#">REV64</a>	-
0		00001	<a href="#">REV16 (vector)</a>	-
0		00010	<a href="#">SADDLP</a>	-
0		00011	<a href="#">SUQADD</a>	-
0		00100	<a href="#">CLS (vector)</a>	-
0		00101	<a href="#">CNT</a>	-
0		00110	<a href="#">SADALP</a>	-
0		00111	<a href="#">SQABS</a>	-
0		01000	<a href="#">CMGT (zero)</a>	-
0		01001	<a href="#">CMEQ (zero)</a>	-

U	Decode fields		Instruction Details	Architecture Version
	size	opcode		
0		01010	<a href="#">CMLT (zero)</a>	-
0		01011	<a href="#">ABS</a>	-
0		10010	<a href="#">XTN, XTN2</a>	-
0		10011	UNALLOCATED	-
0		10100	<a href="#">SQXTN, SQXTN2</a>	-
0	0x	10110	<a href="#">FCVTN, FCVTN2</a>	-
0	0x	10111	<a href="#">FCVTL, FCVTL2</a>	-
0	0x	11000	<a href="#">FRINTN (vector)</a>	-
0	0x	11001	<a href="#">FRINTM (vector)</a>	-
0	0x	11010	<a href="#">FCVTNS (vector)</a>	-
0	0x	11011	<a href="#">FCVTMS (vector)</a>	-
0	0x	11100	<a href="#">FCVTAS (vector)</a>	-
0	0x	11101	<a href="#">SCVTF (vector, integer)</a>	-
0	0x	11110	<a href="#">FRINT32Z (vector)</a>	ARMv8.5
0	0x	11111	<a href="#">FRINT64Z (vector)</a>	ARMv8.5
0	1x	01100	<a href="#">FCMGT (zero)</a>	-
0	1x	01101	<a href="#">FCMEQ (zero)</a>	-
0	1x	01110	<a href="#">FCMLT (zero)</a>	-
0	1x	01111	<a href="#">FABS (vector)</a>	-
0	1x	11000	<a href="#">FRINTP (vector)</a>	-
0	1x	11001	<a href="#">FRINTZ (vector)</a>	-
0	1x	11010	<a href="#">FCVTPS (vector)</a>	-
0	1x	11011	<a href="#">FCVTZS (vector, integer)</a>	-
0	1x	11100	<a href="#">URECPE</a>	-
0	1x	11101	<a href="#">FRECPE</a>	-
0	1x	11111	UNALLOCATED	-
1		00000	<a href="#">REV32 (vector)</a>	-
1		00001	UNALLOCATED	-
1		00010	<a href="#">UADDLP</a>	-
1		00011	<a href="#">USQADD</a>	-
1		00100	<a href="#">CLZ (vector)</a>	-
1		00110	<a href="#">UADALP</a>	-
1		00111	<a href="#">SQNEG</a>	-
1		01000	<a href="#">CMGE (zero)</a>	-
1		01001	<a href="#">CMLE (zero)</a>	-
1		01010	UNALLOCATED	-
1		01011	<a href="#">NEG (vector)</a>	-
1		10010	<a href="#">SQXTUN, SQXTUN2</a>	-
1		10011	<a href="#">SHLL, SHLL2</a>	-
1		10100	<a href="#">UQXTN, UQXTN2</a>	-
1	0x	10110	<a href="#">FCVTXN, FCVTXN2</a>	-
1	0x	10111	UNALLOCATED	-
1	0x	11000	<a href="#">FRINTA (vector)</a>	-
1	0x	11001	<a href="#">FRINTX (vector)</a>	-
1	0x	11010	<a href="#">FCVTNU (vector)</a>	-
1	0x	11011	<a href="#">FCVTMU (vector)</a>	-
1	0x	11100	<a href="#">FCVTAU (vector)</a>	-

Decode fields			Instruction Details	Architecture Version
U	size	opcode		
1	0x	11101	<a href="#">UCVTF (vector, integer)</a>	-
1	0x	11110	<a href="#">FRINT32X (vector)</a>	ARMv8.5
1	0x	11111	<a href="#">FRINT64X (vector)</a>	ARMv8.5
1	00	00101	NOT	-
1	01	00101	RBIT (vector)	-
1	1x	00101	UNALLOCATED	-
1	1x	01100	<a href="#">FCMGE (zero)</a>	-
1	1x	01101	<a href="#">FCMLE (zero)</a>	-
1	1x	01110	UNALLOCATED	-
1	1x	01111	<a href="#">FNEG (vector)</a>	-
1	1x	11000	UNALLOCATED	-
1	1x	11001	<a href="#">FRINTI (vector)</a>	-
1	1x	11010	<a href="#">FCVTPU (vector)</a>	-
1	1x	11011	<a href="#">FCVTZU (vector, integer)</a>	-
1	1x	11100	<a href="#">URSQRTE</a>	-
1	1x	11101	<a href="#">FRSQRTE</a>	-
1	1x	11111	<a href="#">FSQRT (vector)</a>	-

Decode fields			Instruction Details
U	size	opcode	
		1000x	UNALLOCATED
		10101	UNALLOCATED
		11110	UNALLOCATED
	0x	011xx	UNALLOCATED
	0x	11111	UNALLOCATED
	1x	10110	UNALLOCATED
	1x	10111	UNALLOCATED
0		00000	<a href="#">REV64</a>
0		00001	<a href="#">REV16 (vector)</a>
0		00010	<a href="#">SADDLP</a>
0		00011	<a href="#">SUQADD</a>
0		00100	<a href="#">CLS (vector)</a>
0		00101	<a href="#">CNT</a>
0		00110	<a href="#">SADALP</a>
0		00111	<a href="#">SQABS</a>
0		01000	<a href="#">CMGT (zero)</a>
0		01001	<a href="#">CMEQ (zero)</a>
0		01010	<a href="#">CMLT (zero)</a>
0		01011	<a href="#">ABS</a>
0		10010	<a href="#">XTN, XTN2</a>
0		10011	UNALLOCATED
0		10100	<a href="#">SQXTN, SQXTN2</a>
0	0x	10110	<a href="#">FCVTN, FCVTN2</a>
0	0x	10111	<a href="#">FCVTL, FCVTL2</a>
0	0x	11000	<a href="#">FRINTN (vector)</a>
0	0x	11001	<a href="#">FRINTM (vector)</a>
0	0x	11010	<a href="#">FCVTNS (vector)</a>
0	0x	11011	<a href="#">FCVTMS (vector)</a>
0	0x	11100	<a href="#">FCVTAS (vector)</a>



Decode fields			Instruction Details
U	size	opcode	
0	0x	111101	<a href="#">SCVTF (vector, integer)</a>
0	1x	011100	<a href="#">FCMGT (zero)</a>
0	1x	011101	<a href="#">FCMEQ (zero)</a>
0	1x	011110	<a href="#">FCMLT (zero)</a>
0	1x	011111	<a href="#">FABS (vector)</a>
0	1x	110000	<a href="#">FRINTP (vector)</a>
0	1x	110001	<a href="#">FRINTZ (vector)</a>
0	1x	110100	<a href="#">FCVTPS (vector)</a>
0	1x	110111	<a href="#">FCVTZS (vector, integer)</a>
0	1x	111100	<a href="#">URECPE</a>
0	1x	111101	<a href="#">FRECPE</a>
0	1x	111111	UNALLOCATED
1		000000	<a href="#">REV32 (vector)</a>
1		000001	UNALLOCATED
1		000010	<a href="#">UADDLP</a>
1		000011	<a href="#">USQADD</a>
1		000100	<a href="#">CLZ (vector)</a>
1		000110	<a href="#">UADALP</a>
1		000111	<a href="#">SQNEG</a>
1		010000	<a href="#">CMGE (zero)</a>
1		010001	<a href="#">CMLE (zero)</a>
1		010100	UNALLOCATED
1		010111	<a href="#">NEG (vector)</a>
1		100100	<a href="#">SQXTUN, SQXTUN2</a>
1		100111	<a href="#">SHLL, SHLL2</a>
1		101000	<a href="#">UQXTN, UQXTN2</a>
1	0x	101110	<a href="#">FCVTXN, FCVTXN2</a>
1	0x	101111	UNALLOCATED
1	0x	110000	<a href="#">FRINTA (vector)</a>
1	0x	110001	<a href="#">FRINTX (vector)</a>
1	0x	110100	<a href="#">FCVTNU (vector)</a>
1	0x	110111	<a href="#">FCVTMU (vector)</a>
1	0x	111100	<a href="#">FCVTAU (vector)</a>
1	0x	111101	<a href="#">UCVTF (vector, integer)</a>
1	00	001011	<a href="#">NOT</a>
1	01	001011	<a href="#">RBIT (vector)</a>
1	1x	001011	UNALLOCATED
1	1x	011100	<a href="#">FCMGE (zero)</a>
1	1x	011101	<a href="#">FCMLE (zero)</a>
1	1x	011110	UNALLOCATED
1	1x	011111	<a href="#">FNEG (vector)</a>
1	1x	110000	UNALLOCATED
1	1x	110001	<a href="#">FRINTI (vector)</a>
1	1x	110100	<a href="#">FCVTPU (vector)</a>
1	1x	110111	<a href="#">FCVTZU (vector, integer)</a>
1	1x	111100	<a href="#">URSQRTE</a>
1	1x	111101	<a href="#">FRSQRTE</a>

Decode fields			Instruction Details
U	size	opcode	
1	1x	11111	<a href="#">FSQRT (vector)</a>

## Advanced SIMD across lanes

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size	1	1	0	0	0	opcode			1		0	Rn				Rd							

Decode fields			Instruction Details	Architecture Version
U	size	opcode		
		0000x	UNALLOCATED	-
		00010	UNALLOCATED	-
		001xx	UNALLOCATED	-
		0100x	UNALLOCATED	-
		01011	UNALLOCATED	-
		01101	UNALLOCATED	-
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		111xx	UNALLOCATED	-
0		00011	<a href="#">SADDLV</a>	-
0		01010	<a href="#">SMAXV</a>	-
0		11010	<a href="#">SMINV</a>	-
0		11011	<a href="#">ADDV</a>	-
0	00	01100	<a href="#">FMAXNMV</a> — <a href="#">half-precision</a>	ARMv8.2
0	00	01111	<a href="#">FMAXV</a> — <a href="#">half-precision</a>	ARMv8.2
0	01	01100	UNALLOCATED	-
0	01	01111	UNALLOCATED	-
0	10	01100	<a href="#">FMINNMV</a> — <a href="#">half-precision</a>	ARMv8.2
0	10	01111	<a href="#">FMINV</a> — <a href="#">half-precision</a>	ARMv8.2
0	11	01100	UNALLOCATED	-
0	11	01111	UNALLOCATED	-
1		00011	<a href="#">UADDLV</a>	-
1		01010	<a href="#">UMAXV</a>	-
1		11010	<a href="#">UMINV</a>	-
1		11011	UNALLOCATED	-
1	0x	01100	<a href="#">FMAXNMV</a> — <a href="#">single-precision and double-precision</a>	-
1	0x	01111	<a href="#">FMAXV</a> — <a href="#">single-precision and double-precision</a>	-
1	1x	01100	<a href="#">FMINNMV</a> — <a href="#">single-precision and double-precision</a>	-
1	1x	01111	<a href="#">FMINV</a> — <a href="#">single-precision and double-precision</a>	-

## Advanced SIMD three different

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size	1	Rm			opcode			0	0	Rn			Rd										

Decode fields		Instruction Details
U	opcode	
	1111	UNALLOCATED
0	0000	<a href="#">SADDL, SADDL2</a>
0	0001	<a href="#">SADDW, SADDW2</a>
0	0010	<a href="#">SSUBL, SSUBL2</a>
0	0011	<a href="#">SSUBW, SSUBW2</a>
0	0100	<a href="#">ADDHN, ADDHN2</a>
0	0101	<a href="#">SABAL, SABAL2</a>
0	0110	<a href="#">SUBHN, SUBHN2</a>
0	0111	<a href="#">SABDL, SABDL2</a>
0	1000	<a href="#">SMLAL, SMLAL2 (vector)</a>
0	1001	<a href="#">SQDMLAL, SQDMLAL2 (vector)</a>
0	1010	<a href="#">SMLSL, SMLSL2 (vector)</a>
0	1011	<a href="#">SQDMLSL, SQDMLSL2 (vector)</a>
0	1100	<a href="#">SMULL, SMULL2 (vector)</a>
0	1101	<a href="#">SQDMULL, SQDMULL2 (vector)</a>
0	1110	<a href="#">PMULL, PMULL2</a>
1	0000	<a href="#">UADDL, UADDL2</a>
1	0001	<a href="#">UADDW, UADDW2</a>
1	0010	<a href="#">USUBL, USUBL2</a>
1	0011	<a href="#">USUBW, USUBW2</a>
1	0100	<a href="#">RADDHN, RADDHN2</a>
1	0101	<a href="#">UABAL, UABAL2</a>
1	0110	<a href="#">RSUBHN, RSUBHN2</a>
1	0111	<a href="#">UABDL, UABDL2</a>
1	1000	<a href="#">UMLAL, UMLAL2 (vector)</a>
1	1001	UNALLOCATED
1	1010	<a href="#">UMLSL, UMLSL2 (vector)</a>
1	1011	UNALLOCATED
1	1100	<a href="#">UMULL, UMULL2 (vector)</a>
1	1101	UNALLOCATED
1	1110	UNALLOCATED

### Advanced SIMD three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size	1	Rm			opcode				1	Rn			Rd										

Decode fields		Instruction Details	Architecture Version
U	size opcode		
0		00000	<a href="#">SHADD</a>
0		00001	<a href="#">SQADD</a>
0		00010	<a href="#">SRHADD</a>
0		00100	<a href="#">SHSUB</a>
0		00101	<a href="#">SQSUB</a>
0		00110	<a href="#">CMGT (register)</a>
0		00111	<a href="#">CMGE (register)</a>
0		01000	<a href="#">SSHL</a>

U	Decode fields size	opcode	Instruction Details	Architecture Version
0		01001	<a href="#">SQSHL (register)</a>	-
0		01010	<a href="#">SRSHL</a>	-
0		01011	<a href="#">SQRSHL</a>	-
0		01100	<a href="#">SMAX</a>	-
0		01101	<a href="#">SMIN</a>	-
0		01110	<a href="#">SABD</a>	-
0		01111	<a href="#">SABA</a>	-
0		10000	<a href="#">ADD (vector)</a>	-
0		10001	<a href="#">CMTST</a>	-
0		10010	<a href="#">MLA (vector)</a>	-
0		10011	<a href="#">MUL (vector)</a>	-
0		10100	<a href="#">SMAXP</a>	-
0		10101	<a href="#">SMINP</a>	-
0		10110	<a href="#">SQDMULH (vector)</a>	-
0		10111	<a href="#">ADDP (vector)</a>	-
0	0x	11000	<a href="#">FMAXNM (vector)</a>	-
0	0x	11001	<a href="#">FMLA (vector)</a>	-
0	0x	11010	<a href="#">FADD (vector)</a>	-
0	0x	11011	<a href="#">FMULX</a>	-
0	0x	11100	<a href="#">FCMEQ (register)</a>	-
0	0x	1111011101	<del>— FMLAL</del> <a href="#">FMAX (vector)</a> <a href="#">FMLAL, FMLAL2 (vector)</a>	<del>ARMv8.2</del>
0	0x	1111111110	<a href="#">FRECPS</a> <a href="#">FMAX (vector)</a>	-
0	000x	0001111111	<a href="#">AND (vector)</a> <a href="#">FRECPS</a>	-
0	00	1110100011	<a href="#">FMLAL, FMLAL2 (vector)</a> <del>AND (vector)</del> — <a href="#">FMLAL</a>	<del>ARMv8.2</del>
0	01	00011	BIC (vector, register)	-
0	011x	1110111000	<del>FMINNM (vector)</del> <a href="#">UNALLOCATED</a>	-
0	1x	1100011001	<a href="#">FMINNM (vector)</a> <a href="#">FMLS (vector)</a>	-
0	1x	1100111010	<a href="#">FMLS (vector)</a> <a href="#">FSUB (vector)</a>	-
0	1x	1101011011	<del>UNALLOCATED</del> <a href="#">FSUB (vector)</a>	-
0	1x	1101111100	UNALLOCATED	-
0	1x	1110011101	<a href="#">FMLS, FMLS2 (vector)</a> <del>UNALLOCATED</del> — <a href="#">FMLS</a>	<del>ARMv8.2</del>
0	1x	11110	<a href="#">FMIN (vector)</a>	-
0	1x	11111	<a href="#">ERSQRTS</a>	-
0	10	00011	ORR (vector, register)	-
0	1011	1110100011	<a href="#">FMLS, FMLS2 (vector)</a> <a href="#">ORN (vector)</a> — <a href="#">FMLS</a>	<del>ARMv8.2</del>
01	11	0001100000	<a href="#">ORN (vector)</a> <a href="#">UHADD</a>	-
01	11	1110100001	<a href="#">UQADD</a> <a href="#">UNALLOCATED</a>	-
1		0000000010	<a href="#">UHADD</a> <a href="#">URHADD</a>	-
1		0000100100	<a href="#">UQADD</a> <a href="#">UHSUB</a>	-
1		0001000101	<a href="#">URHADD</a> <a href="#">UQSUB</a>	-
1		0010000110	<a href="#">UHSUB</a> <a href="#">CMHI (register)</a>	-
1		0010100111	<a href="#">UQSUB</a> <a href="#">CMHS (register)</a>	-
1		0011001000	<a href="#">CMHI (register)</a> <a href="#">USHL</a>	-
1		0011101001	<a href="#">CMHS (register)</a> <a href="#">UQSHL (register)</a>	-
1		0100001010	<a href="#">USHL</a> <a href="#">URSHL</a>	-
1		0100101011	<a href="#">UQSHL (register)</a> <a href="#">UQRSHL</a>	-
1		0101001100	<a href="#">URSHL</a> <a href="#">UMAX</a>	-

U	Decode fields size	opcode	Instruction Details	Architecture Version
1		0101101101	UQORSHI UMIN	-
1		0110001110	UMAXUABD	-
1		0110101111	UMINUABA	-
1		0111010000	UABDSUB (vector)	-
1		0111110001	UABACMEQ (register)	-
1		1000010010	SUB (vector)MLS (vector)	-
1		1000110011	CMEQ (register)PMUL	-
1		1001010100	MLS (vector)UMAXP	-
1		1001110101	PMUL UMINP	-
1		1010010110	UMAXPSQORDMULH (vector)	-
1		1010110111	UNALLOCATED UMINP	-
1		1011011000	SQORDMULH (vector)FMAXNMP (vector)	-
1		1011111001	FMLAL, FMLAL2 (vector)UNALLOCATED — FMLAL2	ARMv8.2
1	0x	1100011010	FMAXNMP (vector)FADDP (vector)	-
1	0x	1101011011	FADDP (vector)FMUL (vector)	-
1	0x	1101111100	FMUL (vector)FCMGE (register)	-
1	0x	1110011101	FCMGE (register)FACGE	-
1	0x	1110111110	FACGEFMAXP (vector)	-
1	0x	1111011111	FMAXP (vector)FDIV (vector)	-
1	0x00	1111100011	FDIV (vector)EOR (vector)	-
1	0001	00011	EOR (vector)BSL	-
1	001x	1100111000	FMLAL, FMLAL2 (vector)FMINNMP (vector) — FMLAL2	ARMv8.2
1	011x	0001111001	— FMLSL2BSL FMLSL, FMLSL2 (vector)	ARMv8.2
1	011x	1100111010	FABDUNALLOCATED	-
1	1x	1100011011	UNALLOCATEDFMINNMP (vector)	-
1	1x	1101011100	FABDFCMGT (register)	-
1	1x	1101111101	FACGTUNALLOCATED	-
1	1x	1110011110	FCMGT (register)FMINP (vector)	-
1	1x	1110111111	UNALLOCATEDFACGT	-
1	1x10	1111000011	FMINP (vector)BIT	-
1	1x11	1111100011	BIFUNALLOCATED	-
1	10	00011	BIT	!
1	10	11001	FMLSL, FMLSL2 (vector) — FMLSL2	ARMv8.2
1	11	00011	BIF	!
1	11	11001	UNALLOCATED	!

## Advanced SIMD modified immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode				o2	1	d	e	f	g	h	Rd				

Q	Decode fields op	cmode	o2	Instruction Details	Architecture Version
	0	0xxx	1	UNALLOCATED	-
	0	0xx0	0	MOVI — 32-bit shifted immediate	-
	0	0xx1	0	ORR (vector, immediate) — 32-bit	-
	0	10xx	1	UNALLOCATED	-

Q	Decode fields op	cmode	o2	Instruction Details	Architecture Version
	0	10x0	0	<a href="#">MOVI</a> — <a href="#">16-bit shifted immediate</a>	-
	0	10x1	0	<a href="#">ORR (vector, immediate)</a> — <a href="#">16-bit</a>	-
	0	110x	0	<a href="#">MOVI</a> — <a href="#">32-bit shifting ones</a>	-
	0	110x	1	UNALLOCATED	-
	0	1110	0	<a href="#">MOVI</a> — <a href="#">8-bit</a>	-
	0	1110	1	UNALLOCATED	-
	0	1111	0	<a href="#">FMOV (vector, immediate)</a> — <a href="#">single-precision</a>	-
	0	1111	1	<a href="#">FMOV (vector, immediate)</a> — <a href="#">half-precision</a>	ARMv8.2
	1		1	UNALLOCATED	-
	1	0xx0	0	<a href="#">MVNI</a> — <a href="#">32-bit shifted immediate</a>	-
	1	0xx1	0	<a href="#">BIC (vector, immediate)</a> — <a href="#">32-bit</a>	-
	1	10x0	0	<a href="#">MVNI</a> — <a href="#">16-bit shifted immediate</a>	-
	1	10x1	0	<a href="#">BIC (vector, immediate)</a> — <a href="#">16-bit</a>	-
	1	110x	0	<a href="#">MVNI</a> — <a href="#">32-bit shifting ones</a>	-
0	1	1110	0	<a href="#">MOVI</a> — <a href="#">64-bit scalar</a>	-
0	1	1111	0	UNALLOCATED	-
1	1	1110	0	<a href="#">MOVI</a> — <a href="#">64-bit vector</a>	-
1	1	1111	0	<a href="#">FMOV (vector, immediate)</a> — <a href="#">double-precision</a>	-

### Advanced SIMD shift by immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	1	0	!= 0000				immb			opcode					1	Rn				Rd					
immh																															

The following constraints also apply to this encoding: `immh != 0000 && immh != 0000`

Decode fields U	opcode	Instruction Details
	00001	UNALLOCATED
	00011	UNALLOCATED
	00101	UNALLOCATED
	00111	UNALLOCATED
	01001	UNALLOCATED
	01011	UNALLOCATED
	01101	UNALLOCATED
	01111	UNALLOCATED
	10101	UNALLOCATED
	1011x	UNALLOCATED
	110xx	UNALLOCATED
	11101	UNALLOCATED
	11110	UNALLOCATED
0	00000	<a href="#">SSHR</a>
0	00010	<a href="#">SSRA</a>
0	00100	<a href="#">SRSHR</a>
0	00110	<a href="#">SRSRA</a>
0	01000	UNALLOCATED

Decode fields		Instruction Details
U	opcode	
0	01010	<a href="#">SHL</a>
0	01100	UNALLOCATED
0	01110	<a href="#">SQSHL (immediate)</a>
0	10000	<a href="#">SHRN, SHRN2</a>
0	10001	<a href="#">RSHRN, RSHRN2</a>
0	10010	<a href="#">SQSHRN, SQSHRN2</a>
0	10011	<a href="#">SQRSHRN, SQRSHRN2</a>
0	10100	<a href="#">SSHLL, SSHLL2</a>
0	11100	<a href="#">SCVTF (vector, fixed-point)</a>
0	11111	<a href="#">FCVTZS (vector, fixed-point)</a>
1	00000	<a href="#">USHR</a>
1	00010	<a href="#">USRA</a>
1	00100	<a href="#">URSHR</a>
1	00110	<a href="#">URSRA</a>
1	01000	<a href="#">SRI</a>
1	01010	<a href="#">SLI</a>
1	01100	<a href="#">SQSHLU</a>
1	01110	<a href="#">UQSHL (immediate)</a>
1	10000	<a href="#">SQSHRUN, SQSHRUN2</a>
1	10001	<a href="#">SQRSHRUN, SQRSHRUN2</a>
1	10010	<a href="#">UQSHRN, UQSHRN2</a>
1	10011	<a href="#">UQRSHRN, UQRSHRN2</a>
1	10100	<a href="#">USHLL, USHLL2</a>
1	11100	<a href="#">UCVTF (vector, fixed-point)</a>
1	11111	<a href="#">FCVTZU (vector, fixed-point)</a>

### Advanced SIMD vector x indexed element

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	1	size	L	M		Rm		opcode	H	0								Rn							Rd

Decode fields		Instruction Details	Architecture Version
U	size opcode		
	01	1001	UNALLOCATED
0		0010	<a href="#">SMLAL, SMLAL2 (by element)</a>
0		0011	<a href="#">SQDMLAL, SQDMLAL2 (by element)</a>
0		0110	<a href="#">SMLSL, SMLSL2 (by element)</a>
0		0111	<a href="#">SQDMLSL, SQDMLSL2 (by element)</a>
0		1000	<a href="#">MUL (by element)</a>
0		1010	<a href="#">SMULL, SMULL2 (by element)</a>
0		1011	<a href="#">SQDMULL, SQDMULL2 (by element)</a>
0		1100	<a href="#">SQDMULH (by element)</a>
0		1101	<a href="#">SQRDMULH (by element)</a>
0		1110	<a href="#">SDOT (by element)</a>
0		1111	UNALLOCATED
0	0x	0000	UNALLOCATED
0	0x	0100	UNALLOCATED

U	Decode fields size	opcode	Instruction Details	Architecture Version
0	00	0001	<a href="#">FMLA (by element)</a> — <a href="#">half-precision</a>	ARMv8.2
0	00	0101	<a href="#">FMLS (by element)</a> — <a href="#">half-precision</a>	ARMv8.2
0	00	1001	<a href="#">FMUL (by element)</a> — <a href="#">half-precision</a>	ARMv8.2
0	01	0001	UNALLOCATED	-
0	01	0101	UNALLOCATED	-
0	1x	00010000	<a href="#">FMLA (by element)</a> <del>FMLAL, FMLAL2 (by element)</del> — <a href="#">single-precision and double-precision</a> <del>FMLAL</del>	<del>ARMv8.2</del>
0	1x	01010001	<a href="#">FMLS (by element)</a> <del>FMLA (by element)</del> — <a href="#">single-precision and double-precision</a>	-
0	1x	10010100	<a href="#">FMUL (by element)</a> <del>FMLS, FMUL2 (by element)</del> — <a href="#">single-precision and double-precision</a> <del>FMLS</del>	<del>ARMv8.2</del>
0	101x	00000101	<del>FMLAL, FMLAL2 (by element)</del> <a href="#">FMLS (by element)</a> — <del>FMLAL</del> <a href="#">single-precision and double-precision</a>	<del>ARMv8.2</del>
0	101x	01001001	<del>FMLS, FMUL2 (by element)</del> <a href="#">FMUL (by element)</a> — <del>FMLS</del> <a href="#">single-precision and double-precision</a>	<del>ARMv8.2</del>
01	11	0000	<a href="#">MLA (by element)</a> UNALLOCATED	-
01	11	01000010	<del>UMLAL, UMLAL2 (by element)</del> UNALLOCATED	-
1		00000100	<a href="#">MLA (by element)</a> <a href="#">MLS (by element)</a>	-
1		00100110	<a href="#">UMLAL, UMLAL2 (by element)</a> <del>UMLSL, UMUL2 (by element)</del>	-
1		01001010	<a href="#">MLS (by element)</a> <del>UMUL, UMUL2 (by element)</del>	-
1		01101011	UNALLOCATED <del>UMLSL, UMUL2 (by element)</del>	-
1		10101101	<del>UMUL, UMUL2 (by element)</del> <a href="#">SQRDMLAH (by element)</a>	<del>ARMv8.1</del>
1		10111110	<a href="#">UDOT (by element)</a> UNALLOCATED	<del>ARMv8.2</del>
1		11011111	<a href="#">SQRDMLAH (by element)</a> <del>SQRDMLSH (by element)</del>	ARMv8.1
1		11101000	UNALLOCATED <a href="#">UDOT (by element)</a>	<del>ARMv8.2</del>
1		11111100	UNALLOCATED <a href="#">SQRDMLSH (by element)</a>	<del>ARMv8.1</del>
1	0x00	10000001	UNALLOCATED	-
1	0x00	11000011	UNALLOCATED	-
1	00	00010101	UNALLOCATED	-
1	00	00110111	UNALLOCATED	-
1	00	01011001	<a href="#">FMULX (by element)</a> UNALLOCATED — <a href="#">half-precision</a>	<del>ARMv8.2</del>
1	0001	01110xx1	<a href="#">FCMLA (by element)</a> UNALLOCATED	<del>ARMv8.3</del>
1	001x	10010101	<a href="#">FMULX (by element)</a> — UNALLOCATED <a href="#">half-precision</a>	<del>ARMv8.2</del>
1	011x	0xx11000	— <del>FMLAL2</del> <a href="#">FCMLA (by element)</a> <del>FMLAL, FMLAL2 (by element)</del>	<del>ARMv8.3</del> <del>ARMv8.2</del>
1	1x	1001	<a href="#">FMULX (by element)</a> — <a href="#">single-precision and double-precision</a>	-
1	101x	0xx11100	— <del>FMLS, FMUL2 (by element)</del> <a href="#">FCMLA (by element)</a> <del>FMLS, FMUL2 (by element)</del>	<del>ARMv8.3</del> <del>ARMv8.2</del>
1	10	10000xx1	<del>FMLAL, FMLAL2 (by element)</del> <a href="#">FCMLA (by element)</a> — <del>FMLAL2</del>	<del>ARMv8.2</del> <del>ARMv8.3</del>
1	1011	11000001	<del>FMLS, FMUL2 (by element)</del> — UNALLOCATED <a href="#">FMLS</a>	<del>ARMv8.2</del>
1	11	00010011	UNALLOCATED	-
1	11	00110111	UNALLOCATED	-
1	11	0101	UNALLOCATED	-
1	11	0111	UNALLOCATED	-
1	11	1000	UNALLOCATED	-
1	11	1100	UNALLOCATED	-

## Cryptographic three-register, imm2

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm			1	0	imm2	opcode		Rn			Rd									



Decode fields opcode	Instruction Details	Architecture Version
00	<a href="#">SM3TT1A</a>	ARMv8.2
01	<a href="#">SM3TT1B</a>	ARMv8.2
10	<a href="#">SM3TT2A</a>	ARMv8.2
11	<a href="#">SM3TT2B</a>	ARMv8.2

### Cryptographic three-register SHA 512

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1					Rm	1	0	0	0	opcode											

Decode fields O	opcode	Instruction Details	Architecture Version
0	00	<a href="#">SHA512H</a>	ARMv8.2
0	01	<a href="#">SHA512H2</a>	ARMv8.2
0	10	<a href="#">SHA512SU1</a>	ARMv8.2
0	11	<a href="#">RAX1</a>	ARMv8.2
1	00	<a href="#">SM3PARTW1</a>	ARMv8.2
1	01	<a href="#">SM3PARTW2</a>	ARMv8.2
1	10	<a href="#">SM4EKEY</a>	ARMv8.2
1	11	UNALLOCATED	-

### Cryptographic four-register

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	Op0						Rm	0															

Decode fields Op0	Instruction Details	Architecture Version
00	<a href="#">EOR3</a>	ARMv8.2
01	<a href="#">BCAX</a>	ARMv8.2
10	<a href="#">SM3SS1</a>	ARMv8.2
11	UNALLOCATED	-

### Cryptographic two-register SHA 512

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	opcode	Rn						Rd					

Decode fields opcode	Instruction Details	Architecture Version
00	<a href="#">SHA512SU0</a>	ARMv8.2
01	<a href="#">SM4E</a>	ARMv8.2
1x	UNALLOCATED	-

### Conversion between floating-point and fixed-point

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	S	1	1	1	1	0	type	0	rmode	opcode	scale					Rn					Rd									

Decode fields					Instruction Details		Architecture Version
sf	S	type	rmode	opcode	scale		
				1xx		UNALLOCATED	-
			x0	00x		UNALLOCATED	-
			x1	01x		UNALLOCATED	-
			0x	00x		UNALLOCATED	-
			1x	01x		UNALLOCATED	-
		10				UNALLOCATED	-
	1					UNALLOCATED	-
0					0xxxxxx	UNALLOCATED	-
0	0	00	00	010		<a href="#">SCVTF (scalar, fixed-point)</a> — <a href="#">32-bit to single-precision</a>	-
0	0	00	00	011		<a href="#">UCVTF (scalar, fixed-point)</a> — <a href="#">32-bit to single-precision</a>	-
0	0	00	11	000		<a href="#">FCVTZS (scalar, fixed-point)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	11	001		<a href="#">FCVTZU (scalar, fixed-point)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	01	00	010		<a href="#">SCVTF (scalar, fixed-point)</a> — <a href="#">32-bit to double-precision</a>	-
0	0	01	00	011		<a href="#">UCVTF (scalar, fixed-point)</a> — <a href="#">32-bit to double-precision</a>	-
0	0	01	11	000		<a href="#">FCVTZS (scalar, fixed-point)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	11	001		<a href="#">FCVTZU (scalar, fixed-point)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	11	00	010		<a href="#">SCVTF (scalar, fixed-point)</a> — <a href="#">32-bit to half-precision</a>	ARMv8.2
0	0	11	00	011		<a href="#">UCVTF (scalar, fixed-point)</a> — <a href="#">32-bit to half-precision</a>	ARMv8.2
0	0	11	11	000		<a href="#">FCVTZS (scalar, fixed-point)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
0	0	11	11	001		<a href="#">FCVTZU (scalar, fixed-point)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
1	0	00	00	010		<a href="#">SCVTF (scalar, fixed-point)</a> — <a href="#">64-bit to single-precision</a>	-
1	0	00	00	011		<a href="#">UCVTF (scalar, fixed-point)</a> — <a href="#">64-bit to single-precision</a>	-
1	0	00	11	000		<a href="#">FCVTZS (scalar, fixed-point)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	11	001		<a href="#">FCVTZU (scalar, fixed-point)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	01	00	010		<a href="#">SCVTF (scalar, fixed-point)</a> — <a href="#">64-bit to double-precision</a>	-
1	0	01	00	011		<a href="#">UCVTF (scalar, fixed-point)</a> — <a href="#">64-bit to double-precision</a>	-
1	0	01	11	000		<a href="#">FCVTZS (scalar, fixed-point)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	11	001		<a href="#">FCVTZU (scalar, fixed-point)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	11	00	010		<a href="#">SCVTF (scalar, fixed-point)</a> — <a href="#">64-bit to half-precision</a>	ARMv8.2
1	0	11	00	011		<a href="#">UCVTF (scalar, fixed-point)</a> — <a href="#">64-bit to half-precision</a>	ARMv8.2
1	0	11	11	000		<a href="#">FCVTZS (scalar, fixed-point)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2
1	0	11	11	001		<a href="#">FCVTZU (scalar, fixed-point)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2

## Conversion between floating-point and integer

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	S	1	1	1	1	0	type	1	rmode	opcode	0	0	0	0	0	0	Rn					Rd								

sf	S	Decode fields		opcode	Instruction Details	Architecture Version
		type	rmode			
			x1	01x	UNALLOCATED	-
			x1	10x	UNALLOCATED	-
			1x	01x	UNALLOCATED	-
			1x	10x	UNALLOCATED	-
	0	10		0xx	UNALLOCATED	-
	0	10		10x	UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	x1	11x	UNALLOCATED	-
0	0	00	00	000	<a href="#">FCVTNS (scalar)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	00	001	<a href="#">FCVTNU (scalar)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	00	010	<a href="#">SCVTF (scalar, integer)</a> — <a href="#">32-bit to single-precision</a>	-
0	0	00	00	011	<a href="#">UCVTF (scalar, integer)</a> — <a href="#">32-bit to single-precision</a>	-
0	0	00	00	100	<a href="#">FCVTAS (scalar)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	00	101	<a href="#">FCVTAU (scalar)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	00	110	<a href="#">FMOV (general)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	00	111	<a href="#">FMOV (general)</a> — <a href="#">32-bit to single-precision</a>	-
0	0	00	01	000	<a href="#">FCVTPS (scalar)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	01	001	<a href="#">FCVTPU (scalar)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	1x	11x	UNALLOCATED	-
0	0	00	10	000	<a href="#">FCVTMS (scalar)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	10	001	<a href="#">FCVTMU (scalar)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	11	000	<a href="#">FCVTZS (scalar, integer)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	00	11	001	<a href="#">FCVTZU (scalar, integer)</a> — <a href="#">single-precision to 32-bit</a>	-
0	0	01	0x	11x	UNALLOCATED	-
0	0	01	00	000	<a href="#">FCVTNS (scalar)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	00	001	<a href="#">FCVTNU (scalar)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	00	010	<a href="#">SCVTF (scalar, integer)</a> — <a href="#">32-bit to double-precision</a>	-
0	0	01	00	011	<a href="#">UCVTF (scalar, integer)</a> — <a href="#">32-bit to double-precision</a>	-
0	0	01	00	100	<a href="#">FCVTAS (scalar)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	00	101	<a href="#">FCVTAU (scalar)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	01	000	<a href="#">FCVTPS (scalar)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	01	001	<a href="#">FCVTPU (scalar)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	10	000	<a href="#">FCVTMS (scalar)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	10	001	<a href="#">FCVTMU (scalar)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	10	11x	UNALLOCATED	-
0	0	01	11	000	<a href="#">FCVTZS (scalar, integer)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	11	001	<a href="#">FCVTZU (scalar, integer)</a> — <a href="#">double-precision to 32-bit</a>	-
0	0	01	11	110	<a href="#">FJCVTZS</a>	ARMv8.3
0	0	01	11	111	UNALLOCATED	-
0	0	10		11x	UNALLOCATED	-
0	0	11	00	000	<a href="#">FCVTNS (scalar)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
0	0	11	00	001	<a href="#">FCVTNU (scalar)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
0	0	11	00	010	<a href="#">SCVTF (scalar, integer)</a> — <a href="#">32-bit to half-precision</a>	ARMv8.2
0	0	11	00	011	<a href="#">UCVTF (scalar, integer)</a> — <a href="#">32-bit to half-precision</a>	ARMv8.2
0	0	11	00	100	<a href="#">FCVTAS (scalar)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
0	0	11	00	101	<a href="#">FCVTAU (scalar)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
0	0	11	00	110	<a href="#">FMOV (general)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2

sf	S	Decode fields		opcode	Instruction Details	Architecture Version
		type	rmode			
0	0	11	00	111	<a href="#">FMOV (general)</a> — <a href="#">32-bit to half-precision</a>	ARMv8.2
0	0	11	01	000	<a href="#">FCVTPS (scalar)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
0	0	11	01	001	<a href="#">FCVTPU (scalar)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
0	0	11	10	000	<a href="#">FCVTMS (scalar)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
0	0	11	10	001	<a href="#">FCVTMU (scalar)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
0	0	11	11	000	<a href="#">FCVTZS (scalar, integer)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
0	0	11	11	001	<a href="#">FCVTZU (scalar, integer)</a> — <a href="#">half-precision to 32-bit</a>	ARMv8.2
1	0	00		11x	UNALLOCATED	-
1	0	00	00	000	<a href="#">FCVTNS (scalar)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	00	001	<a href="#">FCVTNU (scalar)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	00	010	<a href="#">SCVTF (scalar, integer)</a> — <a href="#">64-bit to single-precision</a>	-
1	0	00	00	011	<a href="#">UCVTF (scalar, integer)</a> — <a href="#">64-bit to single-precision</a>	-
1	0	00	00	100	<a href="#">FCVTAS (scalar)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	00	101	<a href="#">FCVTAU (scalar)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	01	000	<a href="#">FCVTPS (scalar)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	01	001	<a href="#">FCVTPU (scalar)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	10	000	<a href="#">FCVTMS (scalar)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	10	001	<a href="#">FCVTMU (scalar)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	11	000	<a href="#">FCVTZS (scalar, integer)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	11	001	<a href="#">FCVTZU (scalar, integer)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	01	x1	11x	UNALLOCATED	-
1	0	01	00	000	<a href="#">FCVTNS (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	00	001	<a href="#">FCVTNU (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	00	010	<a href="#">SCVTF (scalar, integer)</a> — <a href="#">64-bit to double-precision</a>	-
1	0	01	00	011	<a href="#">UCVTF (scalar, integer)</a> — <a href="#">64-bit to double-precision</a>	-
1	0	01	00	100	<a href="#">FCVTAS (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	00	101	<a href="#">FCVTAU (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	00	110	<a href="#">FMOV (general)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	00	111	<a href="#">FMOV (general)</a> — <a href="#">64-bit to double-precision</a>	-
1	0	01	01	000	<a href="#">FCVTPS (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	01	001	<a href="#">FCVTPU (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	1x	11x	UNALLOCATED	-
1	0	01	10	000	<a href="#">FCVTMS (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	10	001	<a href="#">FCVTMU (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	11	000	<a href="#">FCVTZS (scalar, integer)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	11	001	<a href="#">FCVTZU (scalar, integer)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	10	x0	11x	UNALLOCATED	-
1	0	10	01	110	<a href="#">FMOV (general)</a> — <a href="#">top half of 128-bit to 64-bit</a>	-
1	0	10	01	111	<a href="#">FMOV (general)</a> — <a href="#">64-bit to top half of 128-bit</a>	-
1	0	10	1x	11x	UNALLOCATED	-
1	0	11	00	000	<a href="#">FCVTNS (scalar)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2
1	0	11	00	001	<a href="#">FCVTNU (scalar)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2
1	0	11	00	010	<a href="#">SCVTF (scalar, integer)</a> — <a href="#">64-bit to half-precision</a>	ARMv8.2
1	0	11	00	011	<a href="#">UCVTF (scalar, integer)</a> — <a href="#">64-bit to half-precision</a>	ARMv8.2
1	0	11	00	100	<a href="#">FCVTAS (scalar)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2
1	0	11	00	101	<a href="#">FCVTAU (scalar)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2
1	0	11	00	110	<a href="#">FMOV (general)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2

sf	S	Decode fields		opcode	Instruction Details	Architecture Version
		type	rmode			
1	0	11	00	111	<a href="#">FMOV (general)</a> — <a href="#">64-bit to half-precision</a>	ARMv8.2
1	0	11	01	000	<a href="#">FCVTPS (scalar)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2
1	0	11	01	001	<a href="#">FCVTPU (scalar)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2
1	0	11	10	000	<a href="#">FCVTMS (scalar)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2
1	0	11	10	001	<a href="#">FCVTMU (scalar)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2
1	0	11	11	000	<a href="#">FCVTZS (scalar, integer)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2
1	0	11	11	001	<a href="#">FCVTZU (scalar, integer)</a> — <a href="#">half-precision to 64-bit</a>	ARMv8.2

### Floating-point data-processing (1 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	type	1	opcode					1	0	0	0	0	Rn					Rd						

M	S	Decode fields		opcode	Instruction Details	Architecture Version
		type	rmode			
				1xxxxx1xxxxx	UNALLOCATED	-
	1				UNALLOCATED	-
				1xxxxx	UNALLOCATED	-
0	0	00		000000	<a href="#">FMOV (register)</a> — <a href="#">UNALLOCATED single-precision</a>	-
0	0	00		000001	<a href="#">FABS (scalar)</a> <a href="#">FMOV (register)</a> — <a href="#">single-precision</a>	-
0	0	00		000010	<a href="#">FNEG (scalar)</a> <a href="#">FABS (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00		000011	<a href="#">FSQRT (scalar)</a> <a href="#">FNEG (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00		000100	<a href="#">FSQRT (scalar)</a> UNALLOCATED — <a href="#">single-precision</a>	-
0	0	00		000101	<a href="#">FCVT</a> — <a href="#">UNALLOCATED single-precision to double-precision</a>	-
0	0	00		000110	<a href="#">FCVT</a> UNALLOCATED — <a href="#">single-precision to double-precision</a>	-
0	0	00		000111	<a href="#">FCVT</a> — <a href="#">UNALLOCATED single-precision to half-precision</a>	-
0	0	00		001000	<a href="#">FRINTN (scalar)</a> <a href="#">FCVT</a> — <a href="#">single-precision single-precision to half-precision</a>	-
0	0	00		001001	<a href="#">FRINTP (scalar)</a> <a href="#">FRINTN (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00		001010	<a href="#">FRINTM (scalar)</a> <a href="#">FRINTP (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00		001011	<a href="#">FRINTZ (scalar)</a> <a href="#">FRINTM (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00		001100	<a href="#">FRINTA (scalar)</a> <a href="#">FRINTZ (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00		001101	<a href="#">FRINTA (scalar)</a> UNALLOCATED — <a href="#">single-precision</a>	-
0	0	00		001110	<a href="#">FRINTX (scalar)</a> — <a href="#">UNALLOCATED single-precision</a>	-
0	0	00		001111	<a href="#">FRINTI (scalar)</a> <a href="#">FRINTX (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00		010000	<a href="#">FRINT32Z (scalar)</a> <a href="#">FRINTI (scalar)</a> — <a href="#">single-precision</a>	ARMv8.5
0	0	00	01	010001	<a href="#">FRINT32X (scalar)</a> <a href="#">FMOV (register)</a> — <a href="#">single-precision double-precision</a>	ARMv8.5
0	0	00	01	010010	<a href="#">FRINT64Z (scalar)</a> <a href="#">FABS (scalar)</a> — <a href="#">single-precision double-precision</a>	ARMv8.5
0	0	00	01	010011	<a href="#">FRINT64X (scalar)</a> <a href="#">FNEG (scalar)</a> — <a href="#">single-precision double-precision</a>	ARMv8.5
0	0	00	01	0101xx	<a href="#">FSQRT (scalar)</a> UNALLOCATED — <a href="#">double-precision</a>	-
0	0	00	01	011xxx	<a href="#">FCVT</a> UNALLOCATED — <a href="#">double-precision to single-precision</a>	-
0	0	01		000000	<a href="#">FMOV (register)</a> — <a href="#">UNALLOCATED double-precision</a>	-
0	0	01		000001	<a href="#">FABS (scalar)</a> — <a href="#">UNALLOCATED double-precision</a>	-
0	0	01		000010	<a href="#">FNEG (scalar)</a> <a href="#">FCVT</a> — <a href="#">double-precision double-precision to half-precision</a>	-
0	0	01		000011	<a href="#">FSQRT (scalar)</a> <a href="#">FRINTN (scalar)</a> — <a href="#">double-precision</a>	-

M	S	Decode fields		Instruction Details	Architecture Version
		type	opcode		
0	0	01	000100001001	FCVTFRINTP (scalar) — double-precision to single-precisiondouble-precision	-
0	0	01	000101001010	FRINTM (scalar)UNALLOCATED — double-precision	-
0	0	01	000110001011	FRINTZ (scalar)UNALLOCATED — double-precision	-
0	0	01	000111001100	FCVTFRINTA (scalar) — double-precision to half-precisiondouble-precision	-
0	0	01	001000001101	FRINTN (scalar) — UNALLOCATEDdouble-precision	-
0	0	01	001001001110	FRINTP (scalar)FRINTX (scalar) — double-precision	-
0	0	01	001010001111	FRINTM (scalar)FRINTI (scalar) — double-precision	-
0	0	0110	00101100xxxx	FRINTZ (scalar) — UNALLOCATEDdouble-precision	-
0	0	0111	001100000000	FRINTA (scalar)FMOV (register) — double-precisionhalf-precision	ARMv8.2
0	0	0111	001101000001	FABS (scalar)UNALLOCATED — half-precision	ARMv8.2
0	0	0111	001110000010	FRINTX (scalar)FNEG (scalar) — double-precisionhalf-precision	ARMv8.2
0	0	0111	001111000011	FRINTI (scalar)FSQRT (scalar) — double-precisionhalf-precision	ARMv8.2
0	0	0111	010000000100	FRINT32Z (scalar)FCVT — double-precisionhalf-precision to single-precision	ARMv8.5
0	0	0111	010001000101	FRINT32X (scalar)FCVT — double-precisionhalf-precision to double-precision	ARMv8.5
0	0	0111	01001000011x	FRINT64Z (scalar) — UNALLOCATEDdouble-precision	ARMv8.5
0	0	0111	010011001000	FRINT64X (scalar)FRINTN (scalar) — double-precisionhalf-precision	ARMv8.5ARMv8.2
0	0	0111	0101xx001001	FRINTP (scalar)UNALLOCATED — half-precision	ARMv8.2
0	0	0111	011xxx001010	FRINTM (scalar)UNALLOCATED — half-precision	ARMv8.2
0	0	1011	0xxxxx001011	FRINTZ (scalar)UNALLOCATED — half-precision	ARMv8.2
0	0	11	000000001100	FMOV (register)FRINTA (scalar) — half-precision	ARMv8.2
0	0	11	000001001101	FABS (scalar) — UNALLOCATEDhalf-precision	ARMv8.2
0	0	11	000010001110	FNEG (scalar)FRINTX (scalar) — half-precision	ARMv8.2
0	0	11	000011001111	FSQRT (scalar)FRINTI (scalar) — half-precision	ARMv8.2
01	0	11	000100	FCVT — UNALLOCATEDhalf-precision to single-precision	-
0	0	11	000101	FCVT — half-precision to double-precision	-
0	0	11	00011x	UNALLOCATED	-
0	0	11	001000	FRINTN (scalar) — half-precision	ARMv8.2
0	0	11	001001	FRINTP (scalar) — half-precision	ARMv8.2
0	0	11	001010	FRINTM (scalar) — half-precision	ARMv8.2
0	0	11	001011	FRINTZ (scalar) — half-precision	ARMv8.2
0	0	11	001100	FRINTA (scalar) — half-precision	ARMv8.2
0	0	11	001101	UNALLOCATED	-
0	0	11	001110	FRINTX (scalar) — half-precision	ARMv8.2
0	0	11	001111	FRINTI (scalar) — half-precision	ARMv8.2
0	0	11	01xxxx	UNALLOCATED	-
1				UNALLOCATED	-

## Floating-point compare

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	type	1				Rm		op	1	0	0	0				Rn				opcode2				

		Decode fields		opcode2	Instruction Details	Architecture Version
M	S	type	op			
				xxxx1	UNALLOCATED	-
				xxx1x	UNALLOCATED	-
				xx1xx	UNALLOCATED	-
			x1		UNALLOCATED	-
			1x		UNALLOCATED	-
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	00	00000	<a href="#">FCMP</a>	-
0	0	00	00	01000	<a href="#">FCMP</a>	-
0	0	00	00	10000	<a href="#">FCMPE</a>	-
0	0	00	00	11000	<a href="#">FCMPE</a>	-
0	0	01	00	00000	<a href="#">FCMP</a>	-
0	0	01	00	01000	<a href="#">FCMP</a>	-
0	0	01	00	10000	<a href="#">FCMPE</a>	-
0	0	01	00	11000	<a href="#">FCMPE</a>	-
0	0	11	00	00000	<a href="#">FCMP</a>	ARMv8.2
0	0	11	00	01000	<a href="#">FCMP</a>	ARMv8.2
0	0	11	00	10000	<a href="#">FCMPE</a>	ARMv8.2
0	0	11	00	11000	<a href="#">FCMPE</a>	ARMv8.2
1					UNALLOCATED	-

## Floating-point immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	type	1	imm8								1	0	0	imm5					Rd					

		Decode fields		imm5	Instruction Details	Architecture Version
M	S	type				
				xxxx1	UNALLOCATED	-
				xxx1x	UNALLOCATED	-
				xx1xx	UNALLOCATED	-
				x1xxx	UNALLOCATED	-
				1xxxx	UNALLOCATED	-
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	00000		<a href="#">FMOV (scalar, immediate) — single-precision</a>	-
0	0	01	00000		<a href="#">FMOV (scalar, immediate) — double-precision</a>	-
0	0	11	00000		<a href="#">FMOV (scalar, immediate) — half-precision</a>	ARMv8.2
1					UNALLOCATED	-

## Floating-point conditional compare

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	type	1	Rm				cond				0	1	Rn				op	nzcw						



Decode fields				Instruction Details	Architecture Version
M	S	type	op		
		10		UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	0	<a href="#">FCCMP</a> — <a href="#">single-precision</a>	-
0	0	00	1	<a href="#">FCCMPE</a> — <a href="#">single-precision</a>	-
0	0	01	0	<a href="#">FCCMP</a> — <a href="#">double-precision</a>	-
0	0	01	1	<a href="#">FCCMPE</a> — <a href="#">double-precision</a>	-
0	0	11	0	<a href="#">FCCMP</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	1	<a href="#">FCCMPE</a> — <a href="#">half-precision</a>	ARMv8.2
1				UNALLOCATED	-

## Floating-point data-processing (2 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
M	0	S	1	1	1	1	0	type	1	Rm						opcode					1	0	Rn						Rd			

Decode fields				Instruction Details	Architecture Version
M	S	type	opcode		
			1xx1	UNALLOCATED	-
			1x1x	UNALLOCATED	-
			11xx	UNALLOCATED	-
		10		UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	0000	<a href="#">FMUL (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00	0001	<a href="#">FDIV (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00	0010	<a href="#">FADD (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00	0011	<a href="#">FSUB (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00	0100	<a href="#">FMAX (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00	0101	<a href="#">FMIN (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00	0110	<a href="#">FMAXNM (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00	0111	<a href="#">FMINNM (scalar)</a> — <a href="#">single-precision</a>	-
0	0	00	1000	<a href="#">FNMUL (scalar)</a> — <a href="#">single-precision</a>	-
0	0	01	0000	<a href="#">FMUL (scalar)</a> — <a href="#">double-precision</a>	-
0	0	01	0001	<a href="#">FDIV (scalar)</a> — <a href="#">double-precision</a>	-
0	0	01	0010	<a href="#">FADD (scalar)</a> — <a href="#">double-precision</a>	-
0	0	01	0011	<a href="#">FSUB (scalar)</a> — <a href="#">double-precision</a>	-
0	0	01	0100	<a href="#">FMAX (scalar)</a> — <a href="#">double-precision</a>	-
0	0	01	0101	<a href="#">FMIN (scalar)</a> — <a href="#">double-precision</a>	-
0	0	01	0110	<a href="#">FMAXNM (scalar)</a> — <a href="#">double-precision</a>	-
0	0	01	0111	<a href="#">FMINNM (scalar)</a> — <a href="#">double-precision</a>	-
0	0	01	1000	<a href="#">FNMUL (scalar)</a> — <a href="#">double-precision</a>	-
0	0	11	0000	<a href="#">FMUL (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	0001	<a href="#">FDIV (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	0010	<a href="#">FADD (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	0011	<a href="#">FSUB (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	0100	<a href="#">FMAX (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	0101	<a href="#">FMIN (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	0110	<a href="#">FMAXNM (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2



Decode fields				Instruction Details	Architecture Version
M	S	type	opcode		
0	0	11	0111	<a href="#">FMINNM (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	1000	<a href="#">FNMUL (scalar)</a> — <a href="#">half-precision</a>	ARMv8.2
1				UNALLOCATED	-

### Floating-point conditional select

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	type	1	Rm				cond				1	1	Rn				Rd							

Decode fields			Instruction Details	Architecture Version
M	S	type		
		10	UNALLOCATED	-
	1		UNALLOCATED	-
0	0	00	<a href="#">FCSEL</a> — <a href="#">single-precision</a>	-
0	0	01	<a href="#">FCSEL</a> — <a href="#">double-precision</a>	-
0	0	11	<a href="#">FCSEL</a> — <a href="#">half-precision</a>	ARMv8.2
1			UNALLOCATED	-

### Floating-point data-processing (3 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
M	0	S	1	1	1	1	1	type	o1	Rm				o0	Ra				Rn				Rd												

Decode fields					Instruction Details	Architecture Version
M	S	type	o1	o0		
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	0	0	<a href="#">FMADD</a> — <a href="#">single-precision</a>	-
0	0	00	0	1	<a href="#">FMSUB</a> — <a href="#">single-precision</a>	-
0	0	00	1	0	<a href="#">FNMADD</a> — <a href="#">single-precision</a>	-
0	0	00	1	1	<a href="#">FNMSUB</a> — <a href="#">single-precision</a>	-
0	0	01	0	0	<a href="#">FMADD</a> — <a href="#">double-precision</a>	-
0	0	01	0	1	<a href="#">FMSUB</a> — <a href="#">double-precision</a>	-
0	0	01	1	0	<a href="#">FNMADD</a> — <a href="#">double-precision</a>	-
0	0	01	1	1	<a href="#">FNMSUB</a> — <a href="#">double-precision</a>	-
0	0	11	0	0	<a href="#">FMADD</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	0	1	<a href="#">FMSUB</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	1	0	<a href="#">FNMADD</a> — <a href="#">half-precision</a>	ARMv8.2
0	0	11	1	1	<a href="#">FNMSUB</a> — <a href="#">half-precision</a>	ARMv8.2
1					UNALLOCATED	-

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00bet8\\_rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:2018-06-16T09:0445](#)

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages.

## Pseudocodes

### Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

```
// AArch32.VCRMatch()
// =====

boolean AArch32.VCRMatch(bits(32) vaddress)

if UsingAArch32() && ELUsingAArch32(EL1) && IsZero(vaddress<1:0>) && PSTATE.EL != EL2 then
    // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
    match_word = Zeros(32);

    if vaddress<31:5> == ExcVectorBase()<31:5> then
        if HaveEL(EL3) && !IsSecure() then
            match_word<UInt(vaddress<4:2>) + 24> = '1'; // Non-secure vectors
        else
            match_word<UInt(vaddress<4:2>) + 0> = '1'; // Secure vectors (or no EL3)

    if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() && vaddress<31:5> == MVBAR<31:5> then
        match_word<UInt(vaddress<4:2>) + 8> = '1'; // Monitor vectors

    // Mask out bits not corresponding to vectors.
    if !HaveEL(EL3) then
        mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
    elseif !ELUsingAArch32(EL3) then
        mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
    else
        mask = '11011110':'00000000':'11011100':'11011110';

    match_word = match_word AND DBGVCR AND mask;
    match = !IsZero(match_word);

    // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
    if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);
else
    match = FALSE;

return match;
```

### Library pseudocode for aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// =====

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;
```

## Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =====
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert n <= UInt(DBGDIDR.BRPs);

    enabled = DBGBCR[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                    linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);
        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable\_BPMISMATCHHALF);

    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);
        if !value_mismatch then value_mismatch = ConstrainUnpredictableBool(Unpredictable\_BPMISMATCHHALF);

    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
```



```

// AArch32.BreakpointValueMatch()
// =====
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n, bits(32) vaddress, boolean linked_to)

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n > UInt(DBGDIDR.BRPs) then
    (c, n) = ConstrainUnpredictableInteger(0, UInt(DBGDIDR.BRPs), Unpredictable_BPNOTIMPL);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE,FALSE);

// If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
// call from StateMatch for linking).
if DBGBCR[n].E == '0' then return (FALSE,FALSE);

context_aware = (n >= UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
type = DBGBCR[n].BT;

if ((type IN {'011x','11xx'} && !HaveVirtHostExt()) || // Context matching
    (type == '010x' && HaltOnBreakpointOrWatchpoint()) || // Address mismatch
    (type != '0x0x' && !context_aware) || // Context matching
    (type == '1xxx' && !HaveEL(EL2))) then // EL2 extension
    (c, type) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE,FALSE);
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (type == '0x0x');
mismatch   = (type == '010x');
match_vmid = (type == '10xx');
match_cid1 = (type == 'xx1x');
match_cid2 = (type == '11xx');
linked     = (type == 'xxx1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, of if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return (FALSE,FALSE);

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return (FALSE,FALSE);

// Do the comparison.
if match_addr then
    byte = UInt(vaddress<1:0>);
    assert byte IN {0,2}; // "vaddress" is halfword aligned
    byte_select_match = (DBGBCR[n].BAS<byte> == '1');
    BVR_match = vaddress<31:2> == DBGBVR[n]<31:2> && byte_select_match;
elseif match_cid1 then
    BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
if match_vmid then
    if ELUsingAArch32(EL2) then
        vmid = ZeroExtend(VTTBR.VMID, 16);
        bvr_vmid = ZeroExtend(DBGBXR[n]<7:0>, 16);
    elseif !Have16bitVMID() || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGBXR[n]<7:0>, 16);
    else

```

```

        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGXBVR[n]<15:0>;
        BXVR_match = (EL2Enabled() && PSTATE.EL IN {EL0,EL1} &&
            vmid == bvr_vmid);
    elsif match_cid2 then
        BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
            !ELUsingAArch32(EL2) &&
            DBGXBVR[n]<31:0> == CONTEXTIDR_EL2);

    bvr_match_valid = (match_addr || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

    return (match && !mismatch, !match && mismatch);

```



```

// AArch32.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreakpnt, boolean ispriv)
// "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

// If parameters are set to a reserved type, behaves as either disabled or a defined type
if ((HMC:SSC:PxC) IN {'011xx', '100x0', '101x0', '11010', '11101', '1111x'}) || // Reserved
    (HMC == '0' && PxC == '00' && !isbreakpnt) || // Upr/Svc/Sys
    (SSC IN {'01', '10'} && !HaveEL(EL3)) || // No EL3
    (HMC:SSC:PxC == '11000' && ELUsingAArch32(EL3)) || // AArch64 only
    (HMC:SSC != '000' && HMC:SSC != '111' && !HaveEL(EL3) && !HaveEL(EL2)) || // No EL3/EL2
    (HMC:SSC:PxC == '11100' && !HaveEL(EL2)) then // No EL2
    (c, <HMC, SSC, PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL);
assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
if c == Constraint_DISABLED then return FALSE;
// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

PL2_match = HaveEL(EL2) && HMC == '1';
PL1_match = PxC<0> == '1';
PL0_match = PxC<1> == '1';
SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';

el = PSTATE.EL;

if !ispriv && !isbreakpnt then
    priv_match = PL0_match;
elsif SSU_match then
    priv_match = PSTATE.M IN {M32_User, M32_Svc, M32_System};
else
    case el of
        when EL3 priv_match = PL1_match; // EL3 and EL1 are both PL1
        when EL2 priv_match = PL2_match;
        when EL1 priv_match = PL1_match;
        when EL0 priv_match = PL0_match;

    case SSC of
        when '00' security_state_match = TRUE; // Both
        when '01' security_state_match = !IsSecure(); // Non-secure only
        when '10' security_state_match = IsSecure(); // Secure only
        when '11' security_state_match = TRUE; // Both

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));
        last_ctx_cmp = UInt(DBGDIDR.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCT);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED return FALSE; // Disabled
                when Constraint_NONE linked = FALSE; // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

        if linked then
            vaddress = bits(32) UNKNOWN;
            linked_to = TRUE;
            (linked_match, -) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);

```



## Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

```
// AArch32.GenerateDebugExceptions()
// =====

boolean AArch32.GenerateDebugExceptions()
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());
```

## Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```
// AArch32.GenerateDebugExceptionsFrom()
// =====

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)

    if from == EL0 && !ELStateUsingAArch32(EL1, secure) then
        mask = bit UNKNOWN; // PSTATE.D mask, unused for EL0 case
        return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    if HaveEL(EL3) && secure then
        spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
        if spd<1> == '1' then
            enabled = spd<0> == '1';
        else
            // SPD == 0b01 is reserved, but behaves the same as 0b00.
            enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from != EL2;

    return enabled;
```

## Library pseudocode for aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```
// AArch32.CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch32.CheckForPMUOverflow()

    if !ELUsingAArch32(EL1) then return AArch64.CheckForPMUOverflow();
    pmuirq = PMCR.E == '1' && PMINTENSET<31> == '1' && PMOVSSET<31> == '1';
    for n = 0 to UInt(PMCR.N) - 1
        if HaveEL(EL2) then
            hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
            hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
            E = (if n < UInt(hpmn) then PMCR.E else hpme);
        else
            E = PMCR.E;
        if E == '1' && PMINTENSET<n> == '1' && PMOVSSET<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID\_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn\_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSLR_EL0.)

    return pmuirq;
```

## Library pseudocode for aarch32/debug/pmu/AArch32.CountEvents

```
// AArch32.CountEvents()
// =====
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch32.CountEvents(integer n)
    assert n == 31 || n < UInt(PMCR.N);

    if !ELUsingAArch32(EL1) then return AArch64.CountEvents(n);
    // Event counting is disabled in Debug state
    debug = Halted();

    // In Non-secure state, some counters are reserved for EL2
    if HaveEL(EL2) then
        hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
        hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
        E = if n < UInt(hpmn) || n == 31 then PMCR.E else hpme;
    else
        E = PMCR.E;
    enabled = E == '1' && PMCNTENSET<n> == '1';

    if !IsSecure() then
        // Event counting in Non-secure state is allowed unless all of:
        // * EL2 and the HPMD Extension are implemented
        // * Executing at EL2
        // * PMNx is not reserved for EL2
        // * HDCR.HPMD == 1
        if HaveHPMDExt() && PSTATE.EL == EL2 && (n < UInt(hpmn) || n == 31) then
            hpmd = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMD else HDCR.HPMD;
            prohibited = (hpmd == '1');
        else
            prohibited = FALSE;
    else
        // Event counting in Secure state is prohibited unless any one of:
        // * EL3 is not implemented
        // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
        // * EL3 is using AArch32 and SDCR.SPME == 1
        // * Executing at EL0, and SDER.SUNIDEN == 1.
        spme = (if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME);
        prohibited = HaveEL(EL3) && spme == '0' && (PSTATE.EL != EL0 || SDER.SUNIDEN == '0');

    // The IMPLEMENTATION DEFINED authentication interface might override software controls
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // For the cycle counter, PMCR.DP enables counting when otherwise prohibited
    if prohibited && n == 31 then prohibited = (PMCR.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
    filter = if n == 31 then PMCCFILTR else PMEVTYPER[n];

    P = filter<31>;
    U = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';

    case PSTATE.EL of
        when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
        when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
        when EL2 filtered = (NSH == '0');
        when EL3 filtered = (P == '1');

    return !debug && enabled && !prohibited && !filtered;
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// =====
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    if ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    EDSCR.ERR = '1'; HaveSSBSExt() then PSTATE.SSBS = bits(1) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    EDSCR.ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// =====
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    if[] = bits(32) UNKNOWN; HaveSSBSExt() then PSTATE.SSBS = bits(1) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then
        PSTATE.PAN = '1';
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// =====
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    if[] = bits(32) UNKNOWN; HaveSSBSExt() then PSTATE.SSBS = bits(1) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =====

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)

    bottom = if DBGWVR[n]<2> == '1' then 2 else 3;           // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool\(Unpredictable\_WPMASKANDBAS\);
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1)); MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                // Not contiguous
            byte_select_match = ConstrainUnpredictableBool\(Unpredictable\_WPBASCONTIGUOUS\);
            bottom = 3;                                     // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger\(3, 31, Unpredictable\_RESWPMASK\);
        assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNKNOWN};
        case c of
            when Constraint\_DISABLED return FALSE;           // Disabled
            when Constraint\_NONE mask = 0;                   // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<31:mask> == DBGWVR[n]<31:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool\(Unpredictable\_WPMASKEDBITS\);
    else
        WVR_match = vaddress<31:bottom> == DBGWVR[n]<31:bottom>;

    return WVR_match && byte_select_match;
```

## Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =====
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
                                boolean iswrite)
    assert ELUsingAArch32\(S1TranslationRegime\)();
    assert n <= UInt(DBGDIDR.WRPs);

    // "ispriv" is FALSE for LDRT/STRT instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                    linked, DBGWCR[n].LBN, isbreakpnt, ispriv);

    ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// =====
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
            (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress), fault);
    elsif fault.acctype == AccType\_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception type, FaultRecord fault, bits(32) vaddress)

    exception = ExceptionSyndrome(type);

    d_side = type == Exception\_DataAbort;

    exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = fault.ipaddress.NS;
        exception.ipaddress = ZeroExtend(fault.ipaddress.address);
    else
        exception.ipavalid = FALSE;

    return exception;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// =====

AArch32.CheckPCAlignment()

    bits(32) pc = ThisInstrAddr();
    if (CurrentInstrSet() == InstrSet\_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

    // Generate an Alignment fault Prefetch Abort exception
    vaddress = pc;
    acctype = AccType\_IFETCH;
    iswrite = FALSE;
    secondstage = FALSE;
    AArch32.Abort(vaddress, AArch32.AlignmentFault(acctype, iswrite, secondstage));
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

    // The encoding used in the IFSR or DFSR can be Long-descriptor format or Short-descriptor
    // format. Normally, the current translation table format determines the format. For an abort
    // from Non-secure state to Monitor mode, the IFSR or DFSR uses the Long-descriptor format if
    // any of the following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * The abort is synchronous and either:
    //   - It is taken from Hyp mode.
    //   - It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1';
        if !IsSErrorInterrupt(fault) && !long_format then
            long_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';
    d_side = TRUE;
    if long_format then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType\_IC then
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault\_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';

    d_side = FALSE;
    if long_format then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// =====

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' || IsSecondStage(fault) ||
        (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
        (IsDebugException(fault) && HDCR.TDE == '1')));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRExt.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(ExceptionDataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
```



## Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// =====

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' || IsSecondStage(fault) ||
        (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
        (IsDebugException(fault) && HDCR.TDE == '1')));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0C;
    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        if fault.type == Fault_Alignment then // PC Alignment fault
            exception = ExceptionSyndrome(Exception_PCAalignment);
            exception.vaddress = ThisInstrAddr();
        else
            exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/aborts/BranchTargetException

```
// BranchTargetException
// =====
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)

    route_to_el2 = EL2Enabled() && PSTATE.EL == EL0 && HCR.EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_BranchTarget);
    exception.syndrome<1:0> = PSTATE.BTYPE;
    exception.syndrome<24:2> = Zeros(); // RES0

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch32/exceptions/aborts/EffectiveTCF

```
// EffectiveTCF()
// =====
// Returns the TCF field applied to Tag Check Fails in the given Exception Level

bits(2) EffectiveTCF(bits(2) el)
    if el == EL3 then
        tcf = SCTLR_EL3.TCF;
    elsif el == EL2 then
        tcf = SCTLR_EL2.TCF;
    elsif el == EL1 then
        tcf = SCTLR_EL1.TCF;
    elsif el == EL0 && HCR_EL2.<E2H,TGE> == '11' then
        tcf = SCTLR_EL2.TCF0;
    elsif el == EL0 && HCR_EL2.<E2H,TGE> != '11' then
        tcf = SCTLR_EL1.TCF0;

    return tcf;
```

## Library pseudocode for aarch32/exceptions/aborts/RecordTagCheckFail

```
// RecordTagCheckFail()
// =====
// Records a tag fail exception into the appropriate TCFR_ELx

ReportTagCheckFail(bits(2) el, bit ttbr)
    if el == EL3 then
        assert ttbr == '0';
        TFSR_EL3.TF0 = '1';
    elsif el == EL2 then
        if ttbr == '0' then
            TFSR_EL2.TF0 = '1';
        else
            TFSR_EL2.TF1 = '1';
    elsif el == EL1 then
        if ttbr == '0' then
            TFSR_EL1.TF0 = '1';
        else
            TFSR_EL1.TF1 = '1';
    elsif el == EL0 then
        if ttbr == '0' then
            TFSRE0_EL1.TF0 = '1';
        else
            TFSRE0_EL1.TF1 = '1';
```

## Library pseudocode for aarch32/exceptions/aborts/TagCheckFail

```
// TagCheckFail()
// =====
// Handle a tag check fail condition

TagCheckFail(bits(64) vaddress, boolean iswrite)
    bits(2) tcf = EffectiveTCF(PSTATE.EL);
    if tcf == '01' then
        TagCheckFault(vaddress, iswrite);
    elsif tcf == '10' then
        ReportTagCheckFail(PSTATE.EL, vaddress<55>);
```

## Library pseudocode for aarch32/exceptions/aborts/TagCheckFault

```
// TagCheckFault()
// =====
// Raise a tag check fail exception.

TagCheckFault(bits(64) va, boolean write)
    bits(2) target_el;
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset = 0x0;

    if PSTATE.EL == EL0 then
        target_el = if HCR_EL2.TGE == 0 then EL1 else EL2;
    else
        target_el = PSTATE.EL;

    exception = ExceptionSyndrome(Exception_DataAbort);
    exception.syndrome<5:0> = '010001';
    if write then
        exception.syndrome<6> = '1';
    exception.vaddress = va;

    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch32/exceptions/asynch/AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// =====

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalFIQException();
    route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' || HCR.FMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_FIQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/asynch/AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\);
    if !route_to_aarch64 && EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost\(\));
    if !route_to_aarch64 && HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException\(\);

    route_to_monitor = HaveEL\(EL3\) && SCR.IRQ == '1';
    route_to_hyp = (EL2Enabled\(\) && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome\(Exception\_IRQ\);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode\(M32\_IRQ, preferred\_exception\_return, lr\_offset, vect\_offset\);
```

## Library pseudocode for aarch32/exceptions/asynch/AArch32.TakePhysicalSErrorException

```
// AArch32.TakePhysicalSErrorException()
// =====

AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) errortype,
                                     boolean impdef_syndrome, bits(24) full_syndrome)

    ClearPendingPhysicalSError();
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSErrorException(impdef_syndrome, full_syndrome);

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' || HCR.AMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);
    vaddress = bits(32) UNKNOWN;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(ExceptionDataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// =====

AArch32.TakeVirtualFIQException()
    assert EL2Enabled() && PSTATE.EL IN {EL0, EL1};
    if ELUsingAArch32(EL2) then // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR.TGE == '0' && HCR.FMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// =====

AArch32.TakeVirtualIRQException()
    assert EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1};

    if ELUsingAArch32\(EL2\) then // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\) then AArch64.TakeVirtualIRQException\(\);

    bits(32) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode\(M32\_IRQ, preferred\_exception\_return, lr\_offset, vect\_offset\);
```

## Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualSErrorException

```
// AArch32.TakeVirtualSErrorException()
// =====

AArch32.TakeVirtualSErrorException(bit extflag, bits(2) errortype, boolean impdef_syndrome, bits(24) fault_address)

    assert EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1};
    if ELUsingAArch32\(EL2\) then // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\) then AArch64.TakeVirtualSErrorException\(impdef\_syndrome, fault\_address\);

    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    if HaveRASExt\(\) then
        if ELUsingAArch32\(EL2\) then
            fault = AArch32.AsynchExternalAbort(FALSE, VDFSR.AET, VDFSR.ExT);
        else
            fault = AArch32.AsynchExternalAbort(FALSE, VESR_EL2.AET, VESR_EL2.ExT);
    else
        fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);

    ClearPendingVirtualSError\(\);
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode\(M32\_Abort, preferred\_exception\_return, lr\_offset, vect\_offset\);
```

## Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// =====

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);
    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH;           // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;

    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```

## Library pseudocode for aarch32/exceptions/debug/DebugException

```
constant bits(4) DebugException_Breakpoint = '0001';
constant bits(4) DebugException_BKPT      = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint = '1010';
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.CheckAdvSIMDOrFPRegisterTraps

```
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =====
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

        if (tid0 == '1' && reg == '0000')           // FPSID
        || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then // MVFRx
            if ELUsingAArch32(EL2) then
                AArch32.AArch32SystemAccessTrap(EL2, ThisInstr());
            else
                AArch64.AArch32SystemAccessTrap(EL2, ThisInstr());
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// =====
// Return the Exception Class and Instruction Length fields for reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception type)

    il = if ThisInstrLength() == 32 then '1' else '0';

    case type of
        when Exception Uncategorized          ec = 0x00; il = '1';
        when Exception WFxTrap                ec = 0x01;
        when Exception CP15RTTTrap            ec = 0x03;
        when Exception CP15RRTTrap            ec = 0x04;
        when Exception CP14RTTTrap            ec = 0x05;
        when Exception CP14DTTTrap            ec = 0x06;
        when Exception AdvSIMDFPAccessTrap    ec = 0x07;
        when Exception FPIDTrap               ec = 0x08;
        when Exception PACTrap                ec = 0x09;
        when Exception CP14RRTTrap            ec = 0x0C;
        when Exception BranchTarget           ec = 0x0D;
        when Exception IllegalState           ec = 0x0E; il = '1';
        when Exception SupervisorCall         ec = 0x11;
        when Exception HypervisorCall         ec = 0x12;
        when Exception MonitorCall            ec = 0x13;
        when Exception ERetTrap               ec = 0x1A;
        when Exception InstructionAbort        ec = 0x20; il = '1';
        when Exception PCAlignment            ec = 0x22; il = '1';
        when Exception DataAbort              ec = 0x24;
        when Exception NV2DataAbort            ec = 0x25;
        when Exception FPTrappedException     ec = 0x28;
        otherwise                             Unreachable();

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;

    return (ec,il);
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

```
// AArch32.GeneralExceptionsToAArch64()
// =====
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
```



## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// =====
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception type = exception.type;

    (ec,il) = AArch32.ExceptionClass(type);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if type IN {Exception\_InstructionAbort, Exception\_PCAalignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif type == Exception\_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch32.ResetControlRegisters(boolean cold_reset);
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// =====
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert HighestELUsingAArch32\(\);

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL\(EL3\) then
        AArch32.WriteMode\(M32\_Svc\);
        SCR.NS = '0'; // Secure state
    elseif HaveEL\(EL2\) then
        AArch32.WriteMode\(M32\_Hyp\);
    else
        AArch32.WriteMode\(M32\_Svc\);

    // Reset the CP14 and CP15 registers and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111'; // All asynchronous exceptions masked
    PSTATE.IT = '00000000'; // IT block state reset
    PSTATE.T = SCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
    PSTATE.E = SCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters\(\);
    AArch32.ResetSIMDFPRegisters\(\);
    AArch32.ResetSpecialRegisters\(\);
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv; // IMPLEMENTATION DEFINED reset vector

    if HaveEL\(EL3\) then
        if MVBAR<0> == '1' then // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';
    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    BranchTo(rv, BranchType RESETBranchType UNKNOWN);
```

## Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

```
// ExcVectorBase()
// =====

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then // Hivecs selected, base = 0xFFFFF000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
```

## Library pseudocode for aarch32/exceptions/ieeeefp/AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// =====

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, element, accumulated_exceptions);
    FPEXC.DEX = '1';
    FPEXC.TFV = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    FPEXC<10:8> = '111'; // VECITR is RES1 FPEXC<7,4:0> =

    AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

```
// AArch32.CallHypervisor()
// =====
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCEException(immediate);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// =====
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate)

    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCEException(immediate);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCEException

```
// AArch32.TakeHVCEException()
// =====

AArch32.TakeHVCEException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;

    exception = ExceptionSyndrome(Exception\_HypervisorCall);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCException

```
// AArch32.TakeSMCException()
// =====

AArch32.TakeSMCException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCException

```
// AArch32.TakeSVCException()
// =====

AArch32.TakeSVCException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();
    route_to_hyp = EL2Enabled() && PSTATE.EL == EL0 && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_SupervisorCall);
        exception.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// =====
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return,
                    integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    spsr = GetPSRFromPSTATE();
    if !(exception.type IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    if ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveSSBSExt() then PSTATE.SSBS = HSCTLR.DSSBS;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTIONBranchType_UNKNOWN);

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// =====
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                 integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    spsr = GetPSRFromPSTATE();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    if {} = spsr; HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then
        PSTATE.PAN = '1';
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTIONBranchType_UNKNOWN);

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// =====
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                        integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    spsr = GetPSRFromPSTATE();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    if [] = spsr; HaveSSBSExt() then PSTATE.SSBS = SCTL.R.DSSBS;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTL.R.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTIONBranchType_UNKNOWN);

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.AArch32SystemAccessTrap

```
// AArch32.AArch32SystemAccessTrap()
// =====
// Trapped AArch32 System register access other than due to CPTR_EL2 or CPACR_EL1.

AArch32.AArch32SystemAccessTrap(bits(2) target_el, bits(32) instr)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if !ELUsingAArch32(target_el) || AArch32.GeneralExceptionsToAArch64() then
        AArch64.AArch32SystemAccessTrap(target_el, instr);

    assert target_el IN {EL1, EL2};

    if target_el == EL2 then
        exception = AArch32.AArch32SystemAccessTrapSyndrome(instr);
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.AArch32SystemAccessTrapSyndrome

```
// AArch32.AArch32SystemAccessTrapSyndrome()
// =====
// Return the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.AArch32SystemAccessTrapSyndrome(bits(32) instr)
    ExceptionRecord exception;
    cpnum = UInt(instr<11:8>);

    bits(20) iss = Zeros();
    if instr<27:24> == '1110' && instr<4> == '1' && instr<31:28> != '1111' then
        // MRC/MCR
        case cpnum of
            when 10    exception = ExceptionSyndrome(Exception\_FPIDTrap);
            when 14    exception = ExceptionSyndrome(Exception\_CP14RTTrap);
            when 15    exception = ExceptionSyndrome(Exception\_CP15RTTrap);
            otherwise Unreachable();
        iss<19:17> = instr<7:5>;        // opc2
        iss<16:14> = instr<23:21>;      // opc1
        iss<13:10> = instr<19:16>;      // CRn
        iss<8:5>   = instr<15:12>;      // Rt
        iss<4:1>   = instr<3:0>;        // CRm
    elseif instr<27:21> == '1100010' && instr<31:28> != '1111' then
        // MRRC/MCRR
        case cpnum of
            when 14    exception = ExceptionSyndrome(Exception\_CP14RRTTrap);
            when 15    exception = ExceptionSyndrome(Exception\_CP15RRTTrap);
            otherwise Unreachable();
        iss<19:16> = instr<7:4>;        // opc1
        iss<13:10> = instr<19:16>;      // Rt2
        iss<8:5>   = instr<15:12>;      // Rt
        iss<4:1>   = instr<3:0>;        // CRm
    elseif instr<27:25> == '110' && instr<31:28> != '1111' then
        // LDC/STC
        assert cpnum == 14;
        exception = ExceptionSyndrome(Exception\_CP14DTTrap);
        iss<19:12> = instr<7:0>;        // imm8
        iss<4>     = instr<23>;          // U
        iss<2:1>   = instr<24,21>;      // P,W
        if instr<19:16> == '1111' then  // Rn==15, LDC(Literal addressing)/STC
            iss<8:5> = bits(4) UNKNOWN;
            iss<3>   = '1';
        else
            iss<8:5> = instr<19:16>;    // Rn
            iss<3>   = '0';
    else
        Unreachable();
    iss<0> = instr<20>;                // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0>  = iss;

    return exception;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEEnabled

```
// AArch32.CheckAdvSIMDOrFPEEnabled()
// =====
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEEnabled(boolean fpexc_check, boolean advsimd)
  if PSTATE.EL == EL0 && (!HaveEL\(EL2\)) || (!ELUsingAArch32\(EL2\) && HCR_EL2.TGE == '0')) && !ELUsingAArch32\(EL2\)
    // The PE behaves as if FPEXC.EN is 1
    AArch64.CheckFPAdvSIMDEnabled\(\);
  elseif PSTATE.EL == EL0 && HaveEL\(EL2\) && !ELUsingAArch32\(EL2\) && HCR_EL2.TGE == '1' && !ELUsingAArch32\(EL2\)
    if fpexc_check && HCR_EL2.RW == '0' then
      fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
      if fpexc_en == '0' then UNDEFINED;
      AArch64.CheckFPAdvSIMDEnabled\(\);
    else
      cpacr_asedis = CPACR.ASEDIS;
      cpacr_cp10 = CPACR.cp10;

      if HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && !IsSecure\(\) then
        // Check if access disabled in NSACR
        if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
        if NSACR.cp10 == '0' then cpacr_cp10 = '00';

      if PSTATE.EL != EL2 then
        // Check if Advanced SIMD disabled in CPACR
        if advsimd && cpacr_asedis == '1' then UNDEFINED;

        if cpacr_cp10 == '10' then
          (c, cpacr_cp10) = ConstrainUnpredictableBits\(Unpredictable\_RESCPACR\);

        // Check if access disabled in CPACR
        case cpacr_cp10 of
          when '00' disabled = TRUE;
          when '01' disabled = PSTATE.EL == EL0;
          when '11' disabled = FALSE;
          if disabled then UNDEFINED;

      // If required, check FPEXC enabled bit.
      if fpexc_check && FPEXC.EN == '0' then UNDEFINED;

      AArch32.CheckFPAdvSIMDTrap(advsimd); // Also check against HCPTR and CPTR_EL3
```



## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
  if EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
    AArch64.CheckFPAdvSIMDTrap\(\);
  else
    if HaveEL\(EL2\) && !IsSecure\(\) then
      hcptr_tase = HCPTR.TASE;
      hcptr_cp10 = HCPTR.TCP10;

      if HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && !IsSecure\(\) then
        // Check if access disabled in NSACR
        if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
        if NSACR.cp10 == '0' then hcptr_cp10 = '1';

        // Check if access disabled in HCPTR
        if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
          exception = ExceptionSyndrome\(Exception\_AdvSIMDFPAccessTrap\);
          exception.syndrome<24:20> = ConditionSyndrome\(\);

          if advsimd then
            exception.syndrome<5> = '1';
          else
            exception.syndrome<5> = '0';
            exception.syndrome<3:0> = '1010';           // coproc field, always 0xA

          if PSTATE.EL == EL2 then
            AArch32.TakeUndefInstrException\(exception\);
          else
            AArch32.TakeHypTrapException\(exception\);

    if HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
      // Check if access disabled in CPTR_EL3
      if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap\(EL3\);
  return;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap

```
// AArch32.CheckForSMCUndefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
  if !HaveEL\(EL3\) || PSTATE.EL == EL0 then
    UNDEFINED;

  if EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
    AArch64.CheckForSMCUndefOrTrap\(Zeros\(16\)\);
  else
    route_to_hyp = HaveEL\(EL2\) && !IsSecure\(\) && PSTATE.EL == EL1 && HCR.TSC == '1';
    if route_to_hyp then
      exception = ExceptionSyndrome\(Exception\_MonitorCall\);
      AArch32.TakeHypTrapException\(exception\);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWfxTrap

```
// AArch32.CheckForWfxTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch32.CheckForWfxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        AArch64.CheckForWfxTrap(target_el, is_wfe);
        return;
    case target_el of
        when EL1 trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2 trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3 trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';
    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.WfxTrap(target_el, is_wfe);
        if target_el == EL3 then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            exception = ExceptionSyndrome(Exception WfxTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
            exception.syndrome<0> = if is_wfe then '1' else '0';
            AArch32.TakeHypTrapException(exception);
        else
            AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// =====
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[.].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

    // Otherwise whether the IT block is allowed depends on hwl of the next instruction.
    next_instr = AArch32.MemSingle[NextInstrAddr(), 2, AccType IFETCH, TRUE];

    if next_instr IN {'11xxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxxx',
                     '01001xxxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
        // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
        // taken on the IT instruction or the next instruction. This is not reflected in
        // the pseudocode, which always takes the exception on the IT instruction. This
        // also does not take into account cases where the next instruction is UNPREDICTABLE.
        UNDEFINED;

    return;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

```
// AArch32.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64\(\) then
        AArch64.CheckIllegalState\(\);
    elseif PSTATE.IL == '1' then
        route_to_hyp = EL2Enabled\(\) && PSTATE.EL == EL0 && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr\(\);
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome\(Exception\_IllegalState\);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException\(\);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// =====
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32\(EL1\) then SCTLR.SED else SCTLR[.SED]);
    if setend_disabled == '1' then
        UNDEFINED;

    return;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL\(EL2\) && !IsSecure\(\) && ELUsingAArch32\(EL2\);

    bits(32) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x14;

    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// =====
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL\(EL3\) && ELUsingAArch32\(EL3\);

    bits(32) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet\(\) == InstrSet\_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException()
    exception = ExceptionSyndrome(Exception Uncategorized);
    AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException(ExceptionRecord exception)

    route_to_hyp = EL2Enabled() && PSTATE.EL == EL0 && HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet\_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    elseif route_to_hyp then
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32\_Undef, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.UndefinedFault

```
// AArch32.UndefinedFault()
// =====

AArch32.UndefinedFault()

    if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefinedFault();
    AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/functions/aborts/AArch32.CreateFaultRecord

```
// AArch32.CreateFaultRecord()
// =====

FaultRecord AArch32.CreateFaultRecord(Fault type, bits(40) ipaddress, bits(4) domain,
                                       integer level, AccType acctype, boolean write, bit extflag,
                                       bits(4) debugmoe, bits(2) errortype, boolean secondstage, boolean

    FaultRecord fault;
    fault.type = type;
    if (type != Fault None && PSTATE.EL != EL2 && TTBCR.EAE == '0' && !secondstage && !s2fslwalk &&
        AArch32.DomainValid(type, level)) then
        fault.domain = domain;
    else
        fault.domain = bits(4) UNKNOWN;
    fault.debugmoe = debugmoe;
    fault.errortype = errortype;
    fault.ipaddress.NS = bit UNKNOWN;
    fault.ipaddress.address = ZeroExtend(ipaddress);
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fslwalk = s2fslwalk;

    return fault;
```

## Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid

```
// AArch32.DomainValid()
// =====
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault type, integer level)
    assert type != Fault\_None;

    case type of
        when Fault\_Domain
            return TRUE;
        when Fault\_Translation, Fault\_AccessFlag, Fault\_SyncExternalOnWalk, Fault\_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;
```

## Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusLD

```
// AArch32.FaultStatusLD()
// =====
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
    assert fault.type != Fault\_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType\_DC, AccType\_IC, AccType\_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '1';
    fsr<5:0> = EncodeLDFSC(fault.type, fault.level);

    return fsr;
```

## Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusSD

```
// AArch32.FaultStatusSD()
// =====
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.type != Fault\_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType\_DC, AccType\_IC, AccType\_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '0';
    fsr<10,3:0> = EncodeSDFSC(fault.type, fault.level);
    if d_side then
        fsr<7:4> = fault.domain; // Domain field (data fault only)

    return fsr;
```

## Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

```
// AArch32.FaultSyndrome()
// =====
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.type != Fault\_None;

bits(25) iss = Zeros();
if HaveRASExt() && IsAsyncAbort(fault) then iss<11:10> = fault.errortype; // AET
if d_side then
    if IsSecondStage(fault) && !fault.s2fslwalk then iss<24:14> = LSInstructionSyndrome();
    if fault.acctype IN {AccType\_DC, AccType\_DC\_UNPRIV, AccType\_IC, AccType\_AT} then
        iss<8> = '1'; iss<6> = '1';
    else
        iss<6> = if fault.write then '1' else '0';
if IsExternalAbort(fault) then iss<9> = fault.extflag;
iss<7> = if fault.s2fslwalk then '1' else '0';
iss<5:0> = EncodeLDFSC(fault.type, fault.level);

return iss;
```

## Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =====
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault type, integer level)

bits(5) result;
case type of
  when Fault\_AccessFlag
    assert level IN {1,2};
    result = if level == 1 then '00011' else '00110';
  when Fault\_Alignment
    result = '00001';
  when Fault\_Permission
    assert level IN {1,2};
    result = if level == 1 then '01101' else '01111';
  when Fault\_Domain
    assert level IN {1,2};
    result = if level == 1 then '01001' else '01011';
  when Fault\_Translation
    assert level IN {1,2};
    result = if level == 1 then '00101' else '00111';
  when Fault\_SyncExternal
    result = '01000';
  when Fault\_SyncExternalOnWalk
    assert level IN {1,2};
    result = if level == 1 then '01100' else '01110';
  when Fault\_SyncParity
    result = '11001';
  when Fault\_SyncParityOnWalk
    assert level IN {1,2};
    result = if level == 1 then '11100' else '11110';
  when Fault\_AsyncParity
    result = '11000';
  when Fault\_AsyncExternal
    result = '10110';
  when Fault\_Debug
    result = '00010';
  when Fault\_TLBConflict
    result = '10000';
  when Fault\_Lockdown
    result = '10100';    // IMPLEMENTATION DEFINED
  when Fault\_Exclusive
    result = '10101';    // IMPLEMENTATION DEFINED
  when Fault\_ICacheMaint
    result = '00100';
  otherwise
    Unreachable();

return result;
```

## Library pseudocode for aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// =====

bits(32) A32ExpandImm(bits(12) imm12)

// PSTATE.C argument to following function call does not affect the imm32 result.
(imm32, -) = A32ExpandImm\_C(imm12, PSTATE.C);

return imm32;
```

### Library pseudocode for aarch32/functions/common/A32ExpandImm\_C

```
// A32ExpandImm_C()
// =====

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift\_C(unrotated_value, SRTYPE\_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

### Library pseudocode for aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// =====

(SRTYPE, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when '00'
            shift_t = SRTYPE\_LSL; shift_n = UInt(imm5);
        when '01'
            shift_t = SRTYPE\_LSR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRTYPE\_ASR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRTYPE\_RRX; shift_n = 1;
            else
                shift_t = SRTYPE\_ROR; shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

### Library pseudocode for aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// =====

SRTYPE DecodeRegShift(bits(2) type)
    case type of
        when '00' shift_t = SRTYPE\_LSL;
        when '01' shift_t = SRTYPE\_LSR;
        when '10' shift_t = SRTYPE\_ASR;
        when '11' shift_t = SRTYPE\_ROR;
    return shift_t;
```

### Library pseudocode for aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX\_C(x, carry_in);
    return result;
```

### Library pseudocode for aarch32/functions/common/RRX\_C

```
// RRX_C()
// =====

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```



## Library pseudocode for aarch32/functions/common/SRType

```
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

## Library pseudocode for aarch32/functions/common/Shift

```
// Shift()
// =====

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
    (result, -) = Shift\_C(value, type, amount, carry_in);
    return result;
```

## Library pseudocode for aarch32/functions/common/Shift\_C

```
// Shift_C()
// =====

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType\_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType\_LSL
                (result, carry_out) = LSL\_C(value, amount);
            when SRType\_LSR
                (result, carry_out) = LSR\_C(value, amount);
            when SRType\_ASR
                (result, carry_out) = ASR\_C(value, amount);
            when SRType\_ROR
                (result, carry_out) = ROR\_C(value, amount);
            when SRType\_RRX
                (result, carry_out) = RRX\_C(value, carry_in);

    return (result, carry_out);
```

## Library pseudocode for aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// =====

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm\_C(imm12, PSTATE.C);

    return imm32;
```

## Library pseudocode for aarch32/functions/common/T32ExpandImm\_C

```
// T32ExpandImm_C()
// =====

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR\_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

## Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained CP15 traps in HSTR and HCR.

boolean AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if EL2Enabled() && PSTATE.EL IN {EL0,EL1} then
        if PSTATE.EL == EL0 && !ELUsingAArch32(EL2) then
            return AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !(major IN {4,14}) && HSTR<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR.TIDCP
        if (HCR.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
             (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```



```

// AArch32.CheckSystemAccess()
// =====
// Check System register access instruction for enables and disables

AArch32.CheckSystemAccess(integer cp_num, bits(32) instr)
    assert cp_num == UInt(instr<11:8>) && (cp_num IN {14,15});
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        AArch64.CheckAArch32SystemAccess(instr);
        return;
    // Decode the AArch32 System register access instruction
    if instr<31:28> != '1111' && instr<27:24> == '1110' && instr<4> == '1' then // MRC/MCR
        cpvt = TRUE; cpdt = FALSE; nreg = 1;
        opcl = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn = UInt(instr<19:16>);
        CRm = UInt(instr<3:0>);
    elseif instr<31:28> != '1111' && instr<27:21> == '1100010' then // MRRC/MCRR
        cpvt = TRUE; cpdt = FALSE; nreg = 2;
        opcl = UInt(instr<7:4>);
        CRm = UInt(instr<3:0>);
    elseif instr<31:28> != '1111' && instr<27:25> == '110' && instr<22> == '0' then // LDC/STC
        cpvt = FALSE; cpdt = TRUE; nreg = 0;
        opcl = 0;
        CRn = UInt(instr<15:12>);
    else
        allocated = FALSE;

//
// Coarse-grain decode into CP14 or CP15 encoding space. Each of the CPxxxInstrDecode functions
// returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
if cp_num == 14 then
    // LDC and STC only supported for c5 in CP14 encoding space
    if cpdt && CRn != 5 then
        allocated = FALSE;
    else
        // Coarse-grained decode of CP14 based on opcl field
        case opcl of
            when 0    allocated = CP14DebugInstrDecode(instr);
            when 1    allocated = CP14TraceInstrDecode(instr);
            when 7    allocated = CP14JazelleInstrDecode(instr); // JIDR only
            otherwise allocated = FALSE; // All other values are unallocated

elseif cp_num == 15 then
    // LDC and STC not supported in CP15 encoding space
    if !cpvt then
        allocated = FALSE;
    else
        allocated = CP15InstrDecode(instr);

    // Coarse-grain traps to EL2 have a higher priority than exceptions generated because
    // the access instruction is UNDEFINED
    if AArch32.CheckCP15InstrCoarseTraps(CRn, nreg, CRm) then
        // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
        // User mode is UNDEFINED when the trap is disabled, then it is
        // IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
        // when the trap is enabled.
        if PSTATE.EL == EL0 && EL2Enabled() && !allocated then
            if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0" then
                UNDEFINED;
            AArch32.AArch32SystemAccessTrap(EL2, instr);

else
    allocated = FALSE;

if !allocated then
    UNDEFINED;

// If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
AArch32.CheckSystemAccessTraps(instr);

```

```
return;
```

## Library pseudocode for aarch32/functions/coproc/AArch32.CheckSystemAccessEL1Traps

```
// AArch32.CheckSystemAccessEL1Traps()
// =====
// Check for configurable disables or traps to EL1 or EL2 of a System register
// access instruction.

AArch32.CheckSystemAccessEL1Traps(bits(32) instr)
    assert PSTATE.EL == EL0;

    if ((HaveEL(EL1) && IsSecure() && !ELUsingAArch32(EL1)) || IsInHost()) then
        AArch64.CheckAArch32SystemAccessEL1Traps(instr);
        return;
    trap = FALSE;

    // Decode the AArch32 System register access instruction
    (op, cp_num, opcl, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

    if cp_num == 14 then
        if ((op == SystemAccessType\_RT && opcl == 0 && CRn == 0 && CRm == 5 && opc2 == 0) || // DBGDTRR
            (op == SystemAccessType\_DT && CRn == 5 && opc2 == 0)) then // DBGDTRR
            trap = !Halted() && DBGDSCRext.UDCCdis == '1';

        elseif opcl == 0 then
            trap = DBGDSCRext.UDCCdis == '1';

        elseif opcl == 1 then
            trap = CPACR.TRCDIS == '1';
            if HaveEL(EL3) && ELUsingAArch32(EL3) && NSACR.NSTRCDIS == '1' then
                trap = TRUE;

    elseif cp_num == 15 then
        if ((op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 12 && opc2 == 0) || // PMCR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 12 && opc2 == 1) || // PMCNTEN
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 12 && opc2 == 2) || // PMCNTEN
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 12 && opc2 == 3) || // PMOVSR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 12 && opc2 == 6) || // PMCEID
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 12 && opc2 == 7) || // PMCEID
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 13 && opc2 == 1) || // PMXEVT
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 14 && opc2 == 3) || // PMOVSS
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 14 && CRm >= 12)) then // PMEVTY
            trap = PMUSERENR.EN == '0';

        elseif op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 14 && opc2 == 4 then // PMSW
            trap = PMUSERENR.EN == '0' && PMUSERENR.SW == '0';

        elseif ((op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 13 && opc2 == 0) || // PMO
            (op == SystemAccessType\_RRT && opcl == 0 && CRm == 9)) then // PMO
            trap = PMUSERENR.EN == '0' && (write || PMUSERENR.CR == '0');

        elseif ((op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 13 && opc2 == 2) || // PMX
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 14 && CRm >= 8 && CRm <= 11)) then // PM
            trap = PMUSERENR.EN == '0' && (write || PMUSERENR.ER == '0');

        elseif op == SystemAccessType\_RT && opcl == 0 && CRn == 9 && CRm == 12 && opc2 == 5 then // P
            trap = PMUSERENR.EN == '0' && PMUSERENR.ER == '0';

        elseif op == SystemAccessType\_RT && opcl == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then /
            trap = CNTKCTL.PLOPTEN == '0';

        elseif op == SystemAccessType\_RT && opcl == 0 && CRn == 14 && CRm == 0 && opc2 == 0 then /
            trap = CNTKCTL.PLOPCTEN == '0' && CNTKCTL.PLOVCTEN == '0';

        elseif op == SystemAccessType\_RRT && opcl == 1 && CRm == 14 then /
            trap = CNTKCTL.PLOVCTEN == '0';

    if trap then
        AArch32.AArch32SystemAccessTrap(EL1, instr);
```



```

// AArch32.CheckSystemAccessEL2Traps()
// =====
// Check for configurable traps to EL2 of a System register access instruction.

AArch32.CheckSystemAccessEL2Traps(bits(32) instr)
    assert EL2Enabled() && PSTATE.EL IN {EL0, EL1, EL2};

    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckAArch32SystemAccessEL2Traps(instr);
        return;
    trap = FALSE;

    // Decode the AArch32 System register access instruction
    (op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

    if cp_num == 14 && PSTATE.EL IN {EL0, EL1} then
        if ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 0) || // DBGDRAR
            (op == SystemAccessType\_RRT && opc1 == 0 && CRm == 1) || // DBGDRAR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 0) || // DBGDSAR
            (op == SystemAccessType\_RRT && opc1 == 0 && CRm == 2)) then // DBGDSAR
            trap = HDCR.TDRA == '1' || HDCR.TDE == '1' || HCR.TGE == '1';

        elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 4) || // DBGCRAR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 4) || // DBGCRAR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 4) || // DBGCRAR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 4 && opc2 == 4)) then // DBGCRAR
            trap = HDCR.TDOSA == '1' || HDCR.TDE == '1' || HCR.TGE == '1';

        elseif opc1 == 0 && (!Halted() || !(op == SystemAccessType\_RT && CRn == 0 && CRm == 5 && opc2 == 0)) then
            trap = HDCR.TDA == '1' || HDCR.TDE == '1' || HCR.TGE == '1';

        elseif opc1 == 1 then
            trap = HCPTR.TTA == '1';
            if HaveEL(EL3) && ELUsingAArch32(EL3) && NSACR.NSTRCDIS == '1' then
                trap = TRUE;

        elseif op == SystemAccessType\_RT && opc1 == 7 && CRn == 0 && CRm == 0 && opc2 == 0 then // JIDR
            trap = HCR.TID0 == '1';

    elseif cp_num == 14 && PSTATE.EL == EL2 then
        if opc1 == 1 then
            trap = HCPTR.TTA == '1';

    elseif cp_num == 15 && PSTATE.EL IN {EL0, EL1} then
        if ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 0) || // SCTLR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 0) || // TTBR0
            (op == SystemAccessType\_RRT && opc1 == 0 && CRm == 2) || // TTBR0
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 1) || // TTBR1
            (op == SystemAccessType\_RRT && opc1 == 1 && CRm == 2) || // TTBR1
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 2) || // TTBCR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 3) || // TTBCR2
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 3 && CRm == 0 && opc2 == 0) || // DACR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 5 && CRm == 0 && opc2 == 0) || // DFSR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 5 && CRm == 0 && opc2 == 1) || // IFSR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 6 && CRm == 0 && opc2 == 0) || // DFAR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 6 && CRm == 0 && opc2 == 2) || // IFAR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 5 && CRm == 1 && opc2 == 0) || // ADFS
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 5 && CRm == 1 && opc2 == 1) || // AIFS
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 10 && CRm == 2 && opc2 == 0) || // PRRR/M
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 10 && CRm == 2 && opc2 == 1) || // NMRR/M
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 10 && CRm == 3 && opc2 == 0) || // AMAIR0
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 10 && CRm == 3 && opc2 == 1) || // AMAIR1
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 13 && CRm == 0 && opc2 == 1)) then // CON
            trap = if write then HCR.TVM == '1' else HCR.TRVM == '1';

        elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 8 then // TLBI
            trap = write && HCR.TTLB == '1';

        elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 6 && opc2 == 2) || // DCI
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 10 && opc2 == 2) || // DCC
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 14 && opc2 == 2)) then // DCC
            trap = write && HCR.TSW == '1';

```



```

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 6 && opc2 == 1) || // DCI
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 10 && opc2 == 1) || // DCC
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 14 && opc2 == 1)) then // DCC
    trap = write && HCR.TPC == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 1) || // ICI
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 0) || // ICI
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 1 && opc2 == 0) || // ICI
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 11 && opc2 == 1)) then // DCC
    trap = write && HCR.TPU == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 1) || // AC
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 3)) then // AC
    trap = HCR.TAC == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 2) || // TCM
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 3) || // TLB
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 6) || // REV
        (op == SystemAccessType\_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 7)) then // AID
    trap = HCR.TID1 == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 1) || // CTR
        (op == SystemAccessType\_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 0) || // CCS
        (op == SystemAccessType\_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 2) || // CCS
        (op == SystemAccessType\_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 1) || // CLI
        (op == SystemAccessType\_RT && opc1 == 2 && CRn == 0 && CRm == 0 && opc2 == 0)) then // CSS
    trap = HCR.TID2 == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 1) || // ID_
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 2 && opc2 <= 7) || // ID_
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm >= 3 && opc2 <= 1) || // Res_
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 3 && opc2 == 2) || // Res_
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 IN {4,5})) then //
    trap = HCR.TID3 == '1';

elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 2 then // CPA
    trap = HCPTR.TCPAC == '1';

elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0 then // PMO
    trap = HDCR.TPMCR == '1' || HDCR.TPM == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 14 && CRm >= 8) || // PMEVCNTR<n>
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm IN {12,13,14}) || // PM*
        (op == SystemAccessType\_RRT && opc1 == 0 && CRm == 9)) then // PMCCNTR (MR
    trap = HDCR.TPM == '1';

elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then
    trap = CNTHCTL.PL1PCEN == '0';
elseif op == SystemAccessType\_RRT && opc1 == 0 && CRm == 14 then
    trap = CNTHCTL.PL1PCTEN == '0';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 0) || // S
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 2) || // N
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 12 && CRm == 0 && opc2 == 1) || // M
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 1) || // S
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 8 && opc2 >= 4)) then // A
    trap = IsSecureEL2Enabled() && PSTATE.EL == EL1 && IsSecure() && ELUsingAArch32(EL1);

if trap then
    AArch32.AArch32SystemAccessTrap(EL2, instr);

```

## Library pseudocode for aarch32/functions/coproc/AArch32.CheckSystemAccessTraps

```
// AArch32.CheckSystemAccessTraps()
// =====
// Check for configurable disables or traps to a higher EL of an System register access.

AArch32.CheckSystemAccessTraps(bits(32) instr)

    if PSTATE.EL == EL0 then
        AArch32.CheckSystemAccessEL1Traps(instr);
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1, EL2} && !IsInHost() then
        AArch32.CheckSystemAccessEL2Traps(instr);
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && PSTATE.EL IN {EL0, EL1, EL2} then
        AArch64.CheckAArch32SystemAccessEL3Traps(instr);
```

## Library pseudocode for aarch32/functions/coproc/AArch32.DecodeSysRegAccess

```
// AArch32.DecodeSysRegAccess()
// =====
// Decode an AArch32 System register access instruction into its operands.

(SystemAccessType, integer, integer, integer, integer, integer, boolean) AArch32.DecodeSysRegAccess(bits(32) instr)

    cp_num = UInt(instr<11:8>);

    // Decode the AArch32 System register access instruction
    if instr<31:28> != '1111' && instr<27:24> == '1110' && instr<4> == '1' then // MRC/MCR
        op = SystemAccessType\_RT;
        opc1 = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn = UInt(instr<19:16>);
        CRm = UInt(instr<3:0>);
        write = instr<20> == '0';
    elseif instr<31:28> != '1111' && instr<27:21> == '1100010' then // MRRC/MCRR
        op = SystemAccessType\_RRT;
        opc1 = UInt(instr<7:4>);
        CRm = UInt(instr<3:0>);
        write = instr<20> == '0';
    elseif instr<31:28> != '1111' && instr<27:25> == '110' then // LDC/STC
        op = SystemAccessType\_DT;
        CRn = UInt(instr<15:12>);
        write = instr<20> == '0';

    return (op, cp_num, opc1, CRn, CRm, opc2, write);
```

## Library pseudocode for aarch32/functions/coproc/CP14DebugInstrDecode

```
// Decodes an accepted access to a debug System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14DebugInstrDecode(bits(32) instr);
```

## Library pseudocode for aarch32/functions/coproc/CP14JazelleInstrDecode

```
// Decodes an accepted access to a Jazelle System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14JazelleInstrDecode(bits(32) instr);
```

## Library pseudocode for aarch32/functions/coproc/CP14TraceInstrDecode

```
// Decodes an accepted access to a trace System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14TraceInstrDecode(bits(32) instr);
```

## Library pseudocode for aarch32/functions/coproc/CP15InstrDecode

```
// Decodes an accepted access to a System register in the CP15 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP15InstrDecode(bits(32) instr);
```

## Library pseudocode for aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// =====

// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType\_ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    return passed;
```

## Library pseudocode for aarch32/functions/exclusive/AArch32.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

## Library pseudocode for aarch32/functions/exclusive/AArch32.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```

## Library pseudocode for aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// =====

// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)

    acctype = AccType\_ATOMIC;
    iswrite = FALSE;
    aligned = (address == Align(address, size));
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.address, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

## Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```

## Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEEnabled

```
// CheckAdvSIMDOrVFPEEnabled()
// =====

CheckAdvSIMDOrVFPEEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

## Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// =====

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```

## Library pseudocode for aarch32/functions/float/CheckVFPEEnabled

```
// CheckVFPEEnabled()
// =====

CheckVFPEEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
    return;
```

## Library pseudocode for aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =====

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType\_Infinity);  inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);      zero2 = (type2 == FPType\_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

## Library pseudocode for aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =====

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRTType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType\_Infinity);  inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);      zero2 = (type2 == FPType\_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0');
        result = FPHalvedSub(three, product, fpcr);
    return result;
```

## Library pseudocode for aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =====

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTType\_Infinity);  inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);      zero2 = (type2 == FPTType\_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) two = FPTwo('0');
        result = FPSub(two, product, fpcr);
    return result;
```

## Library pseudocode for aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// =====

FPCRType StandardFPSCRValue()
    return '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '00000000000000000000';
```

## Library pseudocode for aarch32/functions/memory/AArch32.CheckAlignment

```
// AArch32.CheckAlignment()
// =====

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype,
                                boolean iswrite)

    if PSTATE.EL == ELO && !ELUsingAArch32(S1TranslationRegime()) then
        A = SCTLRL.A; //use AArch64 register, when higher Exception level is using AArch64
    elsif PSTATE.EL == EL2 then
        A = HSCTLRL.A;
    else
        A = SCTLRL.A;
    aligned = (address == Align(address, alignment));
    atomic = acctype IN { AccType\_ATOMIC, AccType\_ATOMICRW, };
    ordered = acctype IN { AccType\_ORDEREDATOMIC, AccType\_ORDEREDATOMICRW };
    ordered = acctype IN { AccType\_ORDERED, AccType\_ORDEREDRW, AccType\_LIMITEDORDERED, AccType\_ORDEREDATOMICRW };
    vector = acctype == AccType\_VEC;

    // AccType\_VEC is used for SIMD element alignment checks only
    check = (atomic || ordered || vector || A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

## Library pseudocode for aarch32/functions/memory/AArch32.MemSingle

```
// AArch32.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if accdesc = HaveMTEExtCreateAccessDescriptor() then
        if(acctype);
        value = \_Mem[memaddrdesc, size, accdesc];
        return value;

// AArch32.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes. AccessIsTagChecked(ZeroExtend(address, 64),
    bits(4) ptag = TransformTag(ZeroExtend(address, 64));
    if !CheckTag(memaddrdesc, ptag, iswrite) then
        TagCheckFail(ZeroExtend(address, 64), iswrite);
    value = \_Mem[memaddrdesc, size, accdesc];
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8) v
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if accdesc = HaveMTEExtCreateAccessDescriptor() then
        if AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = TransformTag(ZeroExtend(address, 64));
            if !CheckTag(memaddrdesc, ptag, iswrite) then
                TagCheckFail(ZeroExtend(address, 64), iswrite);
(acctype);
        \_Mem[memaddrdesc, size, accdesc] = value;
    return;
```

### Library pseudocode for aarch32/functions/memory/AddressWithAllocationTag

```
// AddressWithAllocationTag()
// =====
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as â€™0000â€™.

bits(64) AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag = allocation_tag - ('000':address<55>);
    result<59:56> = tag;
    return result;
```

### Library pseudocode for aarch32/functions/memory/AllocationTagFromAddress

```
// AllocationTagFromAddress()
// =====
// Generate a Tag from a 64-bit value containing a Logical Address Tag.
// If access to Allocation Tags is disabled, this function returns â€™0000â€™.

bits(4) AllocationTagFromAddress(bits(64) tagged_address)
    bits(4) logical_tag = tagged_address<59:56>;
    bits(4) tag = logical_tag + ('000':tagged_address<55>);
    return tag;
```

### Library pseudocode for aarch32/functions/memory/CheckTag

```
// CheckTag()
// =====
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean CheckTag(AddressDescriptor memaddrdesc, bits(4) ptag, boolean write)
    if memaddrdesc.memattrs.tagged then
        bits(64) paddress = ZeroExtend(memaddrdesc.paddress.address);
        return ptag == MemTag[paddress];
    else
        return TRUE;
```

### Library pseudocode for aarch32/functions/memory/Hint\_PreloadData

```
Hint_PreloadData(bits(32) address);
```

### Library pseudocode for aarch32/functions/memory/Hint\_PreloadDataForWrite

```
Hint_PreloadDataForWrite(bits(32) address);
```

### Library pseudocode for aarch32/functions/memory/Hint\_PreloadInstr

```
Hint_PreloadInstr(bits(32) address);
```



## Library pseudocode for aarch32/functions/memory/MemA

```
// MemA[] - non-assignment form
// =====

bits(8*size) MemA[bits(32) address, integer size]
    acctype = AccType ATOMIC;
    return Mem\_with\_type[address, size, acctype];

// MemA[] - assignment form
// =====

MemA[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType ATOMIC;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemO

```
// MemO[] - non-assignment form
// =====

bits(8*size) MemO[bits(32) address, integer size]
    acctype = AccType ORDERED;
    return Mem\_with\_type[address, size, acctype];

// MemO[] - assignment form
// =====

MemO[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType ORDERED;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemTag

```
// MemTag[] - non-assignment (read) form
// =====
// Load an Allocation Tag from memory.

bits(4) MemTag[bits(64) address]
  AddressDescriptor memaddrdesc;
  bits(4) value;
  iswrite = FALSE;

  memaddrdesc = AArch64.TranslateAddress(address, AccType NORMAL, iswrite, TRUE, TAG_GRANULE);
  // Check for aborts or debug exceptions
  if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

  // Return the granule tag if tagging is enabled...
  if AllocationTagAccessIsEnabled() then
    return MemTag[memaddrdesc];
  else
    // ...otherwise read tag as zero.
    return '0000';

// MemTag[] - assignment (write) form
// =====
// Store an Allocation Tag to memory.

MemTag[bits(64) address] = bits(4) value
  AddressDescriptor memaddrdesc;
  iswrite = TRUE;

  // Stores of allocation tags must be aligned
  if address != Align(address, TAG_GRANULE) then
    boolean secondstage = FALSE;
    AArch64.Abort(address, AArch64.AlignmentFault(AccType NORMAL, iswrite, secondstage));

  wasaligned = TRUE;
  memaddrdesc = AArch64.TranslateAddress(address, AccType NORMAL, iswrite, wasaligned, TAG_GRANULE);

  // Check for aborts or debug exceptions
  if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

  // Memory array access
  if AllocationTagAccessIsEnabled() then
    _MemTag[memaddrdesc] = value;
```

## Library pseudocode for aarch32/functions/memory/MemU

```
// MemU[] - non-assignment form
// =====

bits(8*size) MemU[bits(32) address, integer size]
  acctype = AccType NORMAL;
  return Mem_with_type[address, size, acctype];

// MemU[] - assignment form
// =====

MemU[bits(32) address, integer size] = bits(8*size) value
  acctype = AccType NORMAL;
  Mem_with_type[address, size, acctype] = value;
  return;
```

## Library pseudocode for aarch32/functions/memory/MemU\_unpriv

```
// MemU_unpriv[] - non-assignment form
// =====

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType UNPRIV;
    return Mem\_with\_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// =====

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType UNPRIV;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/Mem\_with\_type

```
// Mem_with_type[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    integer i;
    boolean iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
    if !aligned then
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned];

    if if ( HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '1') || BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem_with_type[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
    integer i;
    boolean iswrite = TRUE;

    if (HaveNV2Ext() && acctype == AccType_NV2REGISTER) acctype = bits(size*8) value
    boolean iswrite = TRUE;

    if && SCTLR_EL2.EE == '1') || BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    else
        AArch32.MemSingle[address, size, acctype, aligned] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/TransformTag

```
// TransformTag()
// =====
// Apply tag transformation rules.

bits(4) TransformTag(bits(64) vaddr)
    bits(4) vtag = vaddr<59:56>;
    bits(4) tagdelta = ZeroExtend(vaddr<55>);
    bits(4) ptag = vtag + tagdelta;
    return ptag;
```

## Library pseudocode for aarch32/functions/memory/boolean

```
// boolean AccessIsTagChecked()
// =====
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AccessIsTagChecked(bits(64) vaddr, AccType acctype)
    if PSTATE.M<4> == '1' then return FALSE;

    if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
        return FALSE;

    if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') then
        return FALSE;

    if !AllocationTagAccessIsEnabled() then
        return FALSE;

    if acctype IN {AccType_IFETCH, AccType_PTW} then
        return FALSE;

    if acctype == AccType_NV2REGISTER then
        return FALSE;

    if PSTATE.TCO == '1' then
        return FALSE;

    if IsNonTagCheckedInstruction() then
        return FALSE;

    return TRUE;
```

## Library pseudocode for aarch32/functions/ras/AArch32.ESBOperation

```
// AArch32.ESBOperation()
// =====
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\);
    if !route_to_aarch64 && EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
    if !route_to_aarch64 && HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.ESBOperation\(\);
        return;

    route_to_monitor = HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && SCR.EA == '1';
    route_to_hyp = EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} && (HCR.TGE == '1' || HCR.AMO == '1');

    if route_to_monitor then
        target = M32\_Monitor;
    elsif route_to_hyp || PSTATE.M == M32\_Hyp then
        target = M32\_Hyp;
    else
        target = M32\_Abort;

    if IsSecure\(\) then
        mask_active = TRUE;
    elsif target == M32\_Monitor then
        mask_active = SCR.AW == '1' && (!HaveEL\(EL2\) || (HCR.TGE == '0' && HCR.AMO == '0'));
    else
        mask_active = target == M32\_Abort || PSTATE.M == M32\_Hyp;

    mask_set = PSTATE.A == '1';
    (-, el) = ELFromM32(target);
    intdis = Halted\(\) || ExternalDebugInterruptsDisabled(el);
    masked = intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending
    if IsPhysicalSErrorPending\(\) && masked then
        syndrome32 = AArch32.PhysicalSErrorSyndrome\(\);
        DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        ClearPendingPhysicalSError\(\);

    return;
```

## Library pseudocode for aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

```
// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();
```

## Library pseudocode for aarch32/functions/ras/AArch32.ReportDeferredError

```
// AArch32.ReportDeferredError()
// =====
// Return deferred SError syndrome

bits(32) AArch32.ReportDeferredError(bits(2) AET, bit ExT)
    bits(32) target;
    target<31> = '1'; // A
    syndrome = Zeros(16);
    if PSTATE.EL == EL2 then
        syndrome<11:10> = AET; // AET
        syndrome<9> = ExT; // EA
        syndrome<5:0> = '010001'; // DFSC
    else
        syndrome<15:14> = AET; // AET
        syndrome<12> = ExT; // ExT
        syndrome<9> = TTBCR.EAE; // LPAE
        if TTBCR.EAE == '1' then // Long-descriptor format
            syndrome<5:0> = '010001'; // STATUS
        else // Short-descriptor format
            syndrome<10,3:0> = '10110'; // FS
    if HaveAnyAArch64() then
        target<24:0> = ZeroExtend(syndrome); // Any RES0 fields must be set to zero
    else
        target<15:0> = syndrome;
    return target;
```

## Library pseudocode for aarch32/functions/ras/AArch32.SErrorSyndrome

```
type AArch32.SErrorSyndrome is (
    bits(2) AET,
    bit ExT
)
```

## Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

```
// AArch32.vESBOperation()
// =====
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
    assert EL2Enabled() && PSTATE.EL IN {EL0, EL1};

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESBOperation();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a
    // virtual SError interrupt might be pending
    vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR.VA == '1';
    vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        VDISR = AArch32.ReportDeferredError(VDFSR<15:14>, VDFSR<12>);
        HCR.VA = '0'; // Clear pending virtual SError

    return;
```

## Library pseudocode for aarch32/functions/registers/AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// =====

AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32\_User] = bits(32) UNKNOWN;
        Rmode[i, M32\_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32\_Hyp] = bits(32) UNKNOWN;    // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32\_User] = bits(32) UNKNOWN;
        Rmode[i, M32\_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32\_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32\_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32\_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32\_Undef] = bits(32) UNKNOWN;
        if HaveEL(EL3) then Rmode[i, M32\_Monitor] = bits(32) UNKNOWN;

    return;
```

## Library pseudocode for aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```
// AArch32.ResetSIMDFPRegisters()
// =====

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

    return;
```

## Library pseudocode for aarch32/functions/registers/AArch32.ResetSpecialRegisters

```
// AArch32.ResetSpecialRegisters()
// =====

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq = bits(32) UNKNOWN;
    SPSR_irq = bits(32) UNKNOWN;
    SPSR_svc = bits(32) UNKNOWN;
    SPSR_abt = bits(32) UNKNOWN;
    SPSR_und = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

    return;
```

## Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

```
AArch32.ResetSystemRegisters(boolean cold_reset);
```



## Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

```
// ALUExceptionReturn()
// =====

ALUExceptionReturn(bits(32) address)
  if PSTATE.EL == EL2 then
    UNDEFINED;
  elsif PSTATE.M IN {M32\_User,M32\_System} then
    UNPREDICTABLE;          // UNDEFINED or NOP
  else
    AArch32.ExceptionReturn(address, SPSR[]);
```

## Library pseudocode for aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// =====

ALUWritePC(bits(32) address)
  if CurrentInstrSet() == InstrSet\_A32 then
    BXWritePC(address, (address););
  else BranchType\_INDIR;
  else
    BranchWritePC(address, BranchType\_INDIR); (address);;
```

## Library pseudocode for aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// =====

BXWritePC(bits(32) address, BXWritePC(bits(32) address))
  if address<0> == '1' then BranchType branch_type)
  if address<0> == '1' then
    SelectInstrSet(InstrSet\_T32);
    address<0> = '0';
  else
    SelectInstrSet(InstrSet\_A32);
    // For branches to an unaligned PC counter in A32 state, the processor takes the branch
    // and does one of:
    // * Forces the address to be aligned
    // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
    if address<1> == '1' && ConstrainUnpredictableBool(Unpredictable\_A32FORCEALIGNPC) then
      address<1> = '0';
    BranchTo(address, BranchType\_UNKNOWN(address, branch_type););
```

## Library pseudocode for aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// =====

BranchWritePC(bits(32) address, BranchWritePC(bits(32) address))
  if BranchType branch_type)
  if CurrentInstrSet() == InstrSet\_A32 then
    address<1:0> = '00';
  else
    address<0> = '0';
    BranchTo(address, BranchType\_UNKNOWN(address, branch_type););
```

## Library pseudocode for aarch32/functions/registers/D

```
// D[] - non-assignment form
// =====

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    return _V[n DIV 2]<base+63:base>;

// D[] - assignment form
// =====

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    _V[n DIV 2]<base+63:base> = value;
    return;
```

## Library pseudocode for aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// =====

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];
```

## Library pseudocode for aarch32/functions/registers/LR

```
// LR - assignment form
// =====

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// =====

bits(32) LR
    return R[14];
```

## Library pseudocode for aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =====

LoadWritePC(bits(32) address)
    BXWritePC(address, (address), BranchType\_INDIR);
```

## Library pseudocode for aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// =====

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    case n of // Select index by mode:      usr fiq irq svc abt und hyp
        when 8      result = RBankSelect(mode, 8, 24, 8, 8, 8, 8, 8);
        when 9      result = RBankSelect(mode, 9, 25, 9, 9, 9, 9, 9);
        when 10     result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11     result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12     result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13     result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14     result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise   result = n;

    return result;
```

## Library pseudocode for aarch32/functions/registers/Monitor\_mode\_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

## Library pseudocode for aarch32/functions/registers/PC

```
// PC - non-assignment form
// =====

bits(32) PC
    return R[15]; // This includes the offset from AArch32 state
```

## Library pseudocode for aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// =====

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before ARMv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

## Library pseudocode for aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =====

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return \_V[n];

// Q[] - assignment form
// =====

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    \_V[n] = value;
    return;
```

## Library pseudocode for aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// =====

bits(128) Qin[integer n]
  assert n >= 0 && n <= 15;
  return Din[2*n+1]:Din[2*n];
```

## Library pseudocode for aarch32/functions/registers/R

```
// R[] - assignment form
// =====

R[integer n] = bits(32) value
  Rmode[n, PSTATE.M] = value;
  return;

// R[] - non-assignment form
// =====

bits(32) R[integer n]
  if n == 15 then
    offset = (if CurrentInstrSet() == InstrSet\_A32 then 8 else 4);
    return \_PC<31:0> + offset;
  else
    return Rmode[n, PSTATE.M];
```

## Library pseudocode for aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =====

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
  integer svc, integer abt, integer und, integer hyp)

  case mode of
    when M32\_User    result = usr; // User mode
    when M32\_FIQ     result = fiq; // FIQ mode
    when M32\_IRQ     result = irq; // IRQ mode
    when M32\_Svc     result = svc; // Supervisor mode
    when M32\_Abort    result = abt; // Abort mode
    when M32\_Hyp     result = hyp; // Hyp mode
    when M32\_Undef    result = und; // Undefined mode
    when M32\_System  result = usr; // System mode uses User mode registers
    otherwise        Unreachable(); // Monitor mode

  return result;
```

## Library pseudocode for aarch32/functions/registers/Rmode

```
// Rmode[] - non-assignment form
// =====

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32\_Monitor;
    assert !BadMode(mode);

    if mode == M32\_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =====

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32\_Monitor;
    assert !BadMode(mode);

    if mode == M32\_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if !HighestELUsingAArch32() && ConstrainUnpredictableBool(Unpredictable\_ZEROUPPER) then
            _R[LookUpRIndex(n, mode)] = ZeroExtend(value);
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;

return;
```

## Library pseudocode for aarch32/functions/registers/S

```
// S[] - non-assignment form
// =====

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    return _V[n DIV 4]<base+31:base>;

// S[] - assignment form
// =====

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    _V[n DIV 4]<base+31:base> = value;
return;
```

## Library pseudocode for aarch32/functions/registers/SP

```
// SP - assignment form
// =====

SP = bits(32) value
  R[13] = value;
  return;

// SP - non-assignment form
// =====

bits(32) SP
  return R[13];
```

## Library pseudocode for aarch32/functions/registers/\_Dclone

```
array bits(64) _Dclone[0..31];
```

## Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =====

AArch32.ExceptionReturn(bits(32) new_pc, bits(32) spsr)

  SynchronizeContext();

  // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
  // mechanism
  SetPSTATEFromPSR(spsr);
  ClearExclusiveLocal(ProcessorID());
  SendEventLocal();

  if PSTATE.IL == '1' then
    // If the exception return is illegal, PC[1:0] are UNKNOWN
    new_pc<1:0> = bits(2) UNKNOWN;
  else
    // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
    if PSTATE.T == '1' then
      new_pc<0> = '0'; // T32
    else
      new_pc<1:0> = '00'; // A32

  BranchTo(new_pc, BranchType ERETBranchType_UNKNOWN);
```

## Library pseudocode for aarch32/functions/system/AArch32.ExecutingATSlxPInstr

```
// AArch32.ExecutingATSlxPInstr()
// =====
// Return TRUE if current instruction is AT SlCPR/WP

boolean AArch32.ExecutingATSlxPInstr()
  if !HavePrivATExt() then return FALSE;

  instr = ThisInstr();
  if instr<24+:4> == '1110' && instr<8+:4> == '1110' then
    op1 = instr<21+:3>;
    CRn = instr<16+:4>;
    CRm = instr<0+:4>;
    op2 = instr<5+:3>;
    return (op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000', '001'});
  else
    return FALSE;
```

## Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

```
// AArch32.ExecutingCP10or11Instr()
// =====

boolean AArch32.ExecutingCP10or11Instr()
    instr = ThisInstr\(\);
    instr_set = CurrentInstrSet\(\);
    assert instr_set IN {InstrSet\_A32, InstrSet\_T32};

    if instr_set == InstrSet\_A32 then
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
    else // InstrSet_T32
        return (instr<31:28> == '111x' && (instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
```

## Library pseudocode for aarch32/functions/system/AArch32.ExecutingLSMInstr

```
// AArch32.ExecutingLSMInstr()
// =====
// Returns TRUE if processor is executing a Load/Store Multiple instruction

boolean AArch32.ExecutingLSMInstr()
    instr = ThisInstr\(\);
    instr_set = CurrentInstrSet\(\);
    assert instr_set IN {InstrSet\_A32, InstrSet\_T32};

    if instr_set == InstrSet\_A32 then
        return (instr<28+4> != '1111' && instr<25+3> == '100');
    else // InstrSet_T32
        if ThisInstrLength\(\) == 16 then
            return (instr<12+4> == '1100');
        else
            return (instr<25+7> == '1110100' && instr<22> == '0');
```

## Library pseudocode for aarch32/functions/system/AArch32.ITAdvance

```
// AArch32.ITAdvance()
// =====

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

## Library pseudocode for aarch32/functions/system/AArch32.SysRegRead

```
// Read from a 32-bit AArch32 System register and return the register's contents.
bits(32) AArch32.SysRegRead(integer cp_num, bits(32) instr);
```

## Library pseudocode for aarch32/functions/system/AArch32.SysRegRead64

```
// Read from a 64-bit AArch32 System register and return the register's contents.
bits(64) AArch32.SysRegRead64(integer cp_num, bits(32) instr);
```

## Library pseudocode for aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

```
// AArch32.SysRegReadCanWriteAPSR()
// =====
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

## Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite

```
// Write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, bits(32) val);
```

## Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite64

```
// Write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, bits(64) val);
```

## Library pseudocode for aarch32/functions/system/AArch32.WriteMode

```
// AArch32.WriteMode()
// =====
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M   = mode;
    PSTATE.EL  = el;
    PSTATE.nRW = '1';
    PSTATE.SP  = (if mode IN {M32\_User,M32\_System} then '0' else '1');
    return;
```



## Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

```
// AArch32.WriteModeByInstr()
// =====
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);

    // 'valid' is set to FALSE if 'mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32\_Hyp || mode == M32\_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32\_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```

## Library pseudocode for aarch32/functions/system/BadMode

```
// BadMode()
// =====

boolean BadMode(bits(5) mode)
    // Return TRUE if 'mode' encodes a mode that is not valid for this implementation
    case mode of
        when M32\_Monitor
            valid = HaveAArch32EL(EL3);
        when M32\_Hyp
            valid = HaveAArch32EL(EL2);
        when M32\_FIQ, M32\_IRQ, M32\_Svc, M32\_Abort, M32\_Undef, M32\_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
            valid = HaveAArch32EL(EL1);
        when M32\_User
            valid = HaveAArch32EL(EL0);
        otherwise
            valid = FALSE; // Passed an illegal mode value
    return !valid;
```

## Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

case SYSm of
  when '000xx', '00100' // R8_usr to R12_usr
    if mode != M32\_FIQ then UNPREDICTABLE;
  when '00101' // SP_usr
    if mode == M32\_System then UNPREDICTABLE;
  when '00110' // LR_usr
    if mode IN {M32\_Hyp, M32\_System} then UNPREDICTABLE;
  when '010xx', '0110x', '01110' // R8_fiq to R12_fiq, SP_fiq, LR_fiq
    if mode == M32\_FIQ then UNPREDICTABLE;
  when '1000x' // LR_irq, SP_irq
    if mode == M32\_IRQ then UNPREDICTABLE;
  when '1001x' // LR_svc, SP_svc
    if mode == M32\_Svc then UNPREDICTABLE;
  when '1010x' // LR_abt, SP_abt
    if mode == M32\_Abort then UNPREDICTABLE;
  when '1011x' // LR_und, SP_und
    if mode == M32\_Undef then UNPREDICTABLE;
  when '1110x' // LR_mon, SP_mon
    if !HaveEL\(EL3\) || !IsSecure\(\) || mode == M32\_Monitor then UNPREDICTABLE;
  when '11110' // ELR_hyp, only from Monitor or Hyp mode
    if !HaveEL\(EL2\) || !(mode IN {M32\_Monitor, M32\_Hyp}) then UNPREDICTABLE;
  when '11111' // SP_hyp, only from Monitor mode
    if !HaveEL\(EL2\) || mode != M32\_Monitor then UNPREDICTABLE;
  otherwise
    UNPREDICTABLE;

return;
```

## Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// =====
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0; // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        // Bit <23> is RES0
        if privileged then
            PSTATE.PAN = value<22>;
        // Bits <21:20> are RES0
        PSTATE.GE = value<19:16>;
    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>; // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);

    return;
```

## Library pseudocode for aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =====

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

## Library pseudocode for aarch32/functions/system/CurrentCond

```
bits(4) AArch32.CurrentCond();
```

## Library pseudocode for aarch32/functions/system/InITBlock

```
// InITBlock()
// =====

boolean InITBlock()
    if CurrentInstrSet() == InstrSet T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

## Library pseudocode for aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// =====

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

## Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// =====

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    new_spsr = SPSR[];

    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>; // N,Z,C,V,Q flags, IT[1:0],J bits

    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>; // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>; // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>; // I,F interrupt masks, T bit, Mode bits

    SPSR[] = new_spsr; // UNPREDICTABLE if User or System mode

    return;
```

## Library pseudocode for aarch32/functions/system/SPSRaccessValid

```
// SPSRaccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110' // SPSR_fiq
            if mode == M32\_FIQ then UNPREDICTABLE;
        when '10000' // SPSR_irq
            if mode == M32\_IRQ then UNPREDICTABLE;
        when '10010' // SPSR_svc
            if mode == M32\_Svc then UNPREDICTABLE;
        when '10100' // SPSR_abt
            if mode == M32\_Abort then UNPREDICTABLE;
        when '10110' // SPSR_und
            if mode == M32\_Undef then UNPREDICTABLE;
        when '11100' // SPSR_mon
            if !HaveEL(EL3) || mode == M32\_Monitor || !IsSecure() then UNPREDICTABLE;
        when '11110' // SPSR_hyp
            if !HaveEL(EL2) || mode != M32\_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

## Library pseudocode for aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// =====

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet\_A32, InstrSet\_T32};
    assert iset IN {InstrSet\_A32, InstrSet\_T32};

    PSTATE.T = if iset == InstrSet\_A32 then '0' else '1';

    return;
```

### Library pseudocode for aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

### Library pseudocode for aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// =====

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```

### Library pseudocode for aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =====

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

## Library pseudocode for aarch32/translation/attrs/AArch32.DefaultTEXDecode

```
// AArch32.DefaultTEXDecode()
// =====

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

MemoryAttributes memattrs;

// Reserved values map to allocated values
if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
    bits(5) texcb;
    (-, texcb) = ConstrainUnpredictableBits(Unpredictable\_RESTEXCB);
    TEX = texcb<4:2>; C = texcb<1>; B = texcb<0>;

case TEX:C:B of
    when '00000'
        // Device-nGnRnE
        memattrs.type = MemType\_Device;
        memattrs.device = DeviceType\_nGnRnE;
    when '00001', '01000'
        // Device-nGnRE
        memattrs.type = MemType\_Device;
        memattrs.device = DeviceType\_nGnRE;
    when '00010', '00011', '00100'
        // Write-back or Write-through Read allocate, or Non-cacheable
        memattrs.type = MemType\_Normal;
        memattrs.inner = ShortConvertAttrsHints(C:B, acctype, FALSE);
        memattrs.outer = ShortConvertAttrsHints(C:B, acctype, FALSE);
        memattrs.shareable = (S == '1');
    when '00110'
        memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    when '00111'
        // Write-back Read and Write allocate
        memattrs.type = MemType\_Normal;
        memattrs.inner = ShortConvertAttrsHints('01', acctype, FALSE);
        memattrs.outer = ShortConvertAttrsHints('01', acctype, FALSE);
        memattrs.shareable = (S == '1');
    when '1xxxx'
        // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
        memattrs.type = MemType\_Normal;
        memattrs.inner = ShortConvertAttrsHints(C:B, acctype, FALSE);
        memattrs.outer = ShortConvertAttrsHints(TEX<1:0>, acctype, FALSE);
        memattrs.shareable = (S == '1');
    otherwise
        // Reserved, handled above
        Unreachable();

// transient bits are not supported in this format
memattrs.inner.transient = FALSE;
memattrs.outer.transient = FALSE;

// distinction between inner and outer shareable is not supported in this format
memattrs.outershareable = memattrs.shareable;
memattrs.tagged = FALSE;

return MemAttrDefaults(memattrs);
```

## Library pseudocode for aarch32/translation/attrs/AArch32.InstructionDevice

```
// AArch32.InstructionDevice()
// =====
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch32.InstructionDevice(AddressDescriptor addrdesc, bits(32) vaddress,
                                           bits(40) ipaddress, integer level, bits(4) domain,
                                           AccType acctype, boolean iswrite, boolean secondstage,
                                           boolean s2fslwalk)

c = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
assert c IN {Constraint\_NONE, Constraint\_FAULT};

if c == Constraint\_FAULT then
    addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, level, acctype, iswrite,
                                             secondstage, s2fslwalk);
else
    addrdesc.memattrs.type = MemType\_Normal;
    addrdesc.memattrs.inner.attrs = MemAttr\_NC;
    addrdesc.memattrs.inner.hints = MemHint\_No;
    addrdesc.memattrs.outer = addrdesc.memattrs.inner;
    addrdesc.memattrs.tagged = FALSE;
    addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

return addrdesc;
```

## Library pseudocode for aarch32/translation/attrs/AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// =====

MemoryAttributes AArch32.RemappedTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

MemoryAttributes memattrs;

region = UInt(TEX<0>:C:B);          // TEX<2:1> are ignored in this mapping scheme
if region == 6 then
    memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
else
    base = 2 * region;
    attrfield = PRRR<base+1:base>;

    if attrfield == '11' then        // Reserved, maps to allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable\_RESPRR);

    case attrfield of
        when '00'                    // Device-nGnRnE
            memattrs.type = MemType\_Device;
            memattrs.device = DeviceType\_nGnRnE;
        when '01'                    // Device-nGnRE
            memattrs.type = MemType\_Device;
            memattrs.device = DeviceType\_nGnRE;
        when '10'
            memattrs.type = MemType\_Normal;
            memattrs.inner = ShortConvertAttrsHints(NMRR<base+1:base>, acctype, FALSE);
            memattrs.outer = ShortConvertAttrsHints(NMRR<base+17:base+16>, acctype, FALSE);
            s_bit = if S == '0' then PRRR.NS0 else PRRR.NS1;
            memattrs.shareable = (s_bit == '1');
            memattrs.outershareable = (s_bit == '1' && PRRR<region+24> == '0');
        when '11'
            Unreachable();

    // transient bits are not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tagged = FALSE;

    return MemAttrDefaults(memattrs);
```



## Library pseudocode for aarch32/translation/attrs/AArch32.S1AttrDecode

```
// AArch32.S1AttrDecode()
// =====
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch32.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;

    if PSTATE.EL == EL2 then
        mair = HMAIR1:HMAIR0;
    else
        mair = MAIR1:MAIR0;
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    memattrs.tagged = FALSE;
    if ((attrfield<7:4> != '0000' && attrfield<7:4> != '1111' && attrfield<3:0> == '0000') ||
if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable\_RESMAIR);
if !
if attrfield<7:4> == '0000' then // Device
memattrs.type = HaveMTEExt() && attrfield<7:4> == '1111' && attrfield<3:0> == '0000' then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable\_RESMAIR);

    if attrfield<7:4> == '0000' then // Device
        memattrs.type = MemType\_Device;
        case attrfield<3:0> of
            when '0000' memattrs.device = DeviceType\_nGnRnE;
            when '0100' memattrs.device = DeviceType\_nGnRE;
            when '1000' memattrs.device = DeviceType\_nGRE;
            when '1100' memattrs.device = DeviceType\_GRE;
            otherwise Unreachable(); // Reserved, handled above

    elsif attrfield<3:0> != '0000' then // Normal
        memattrs.type = MemType\_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    elsif HaveMTEExt() && attrfield == '11110000' then // Tagged, Normal
        memattrs.tagged = TRUE;
        memattrs.type = MemType\_Normal;
        memattrs.outer.attrs = MemAttr\_WB;
        memattrs.inner.attrs = MemAttr\_WB;
        memattrs.outer.hints = MemHint\_RWA;
        memattrs.inner.hints = MemHint\_RWA;
{attrfield<3:0>, acctype};
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';

    else
        Unreachable(); // Reserved, handled above

    return MemAttrDefaults(memattrs);
```

## Library pseudocode for aarch32/translation/attrs/AArch32.TranslateAddressS1Off

```
// AArch32.TranslateAddressS1Off()
// =====
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch32.TranslateAddressS1Off(bits(32) vaddress, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    default_cacheable = (HasS2Translation() && ((if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC) ==

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.type = MemType Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr WB; // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
        result.addrdesc.memattrs.tagged = HCR_EL2.DCT == '1';
    elseif acctype != AccType IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.type = MemType Device;
        result.addrdesc.memattrs.device = DeviceType nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
        result.addrdesc.memattrs.tagged = FALSE;
    else
        // Instruction cacheability controlled by SCTLR/HSCTLR.I
        if PSTATE.EL == EL2 then
            cacheable = HSCTLR.I == '1';
        else
            cacheable = SCTLR.I == '1';
        result.addrdesc.memattrs.type = MemType Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr WT;
            result.addrdesc.memattrs.inner.hints = MemHint RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr NC;
            result.addrdesc.memattrs.inner.hints = MemHint No;
            result.addrdesc.memattrs.shareable = TRUE;
            result.addrdesc.memattrs.outershareable = TRUE;
            result.addrdesc.memattrs.tagged = FALSE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.address = ZeroExtend(vaddress);
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch32.NoFault();
    return result;
```

## Library pseudocode for aarch32/translation/checks/AArch32.AccessIsPrivileged

```
// AArch32.AccessIsPrivileged()
// =====

boolean AArch32.AccessIsPrivileged(AccType acctype)

    el = el = PSTATE.EL;
    if el == AArch32.AccessUsesEL(acctype);

    if el == EL0 then
        ispriv = FALSE;
    elsif el != EL1 then
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType\_UNPRIV);

    return ispriv;
```

## Library pseudocode for aarch32/translation/checks/AArch32.AccessUsesEL

```
// AArch32.AccessUsesEL()
// =====
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch32.AccessUsesEL(AccType acctype)
    if acctype == AccType\_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;
```

## Library pseudocode for aarch32/translation/checks/AArch32.CheckDomain

```
// AArch32.CheckDomain()
// =====

(boolean, FaultRecord) AArch32.CheckDomain(bits(4) domain, bits(32) vaddress, integer level,
AccType acctype, boolean iswrite)

    index = 2 * UInt(domain);
    attrfield = DACR<index+1:index>;

    if attrfield == '10' then // Reserved, maps to an allocated value
        // Reserved value maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable\_RESDACR);

    if attrfield == '00' then
        fault = AArch32.DomainFault(domain, level, acctype, iswrite);
    else
        fault = AArch32.NoFault();

    permissioncheck = (attrfield == '01');

    return (permissioncheck, fault);
```



```

// AArch32.CheckPermission()
// =====
// Function used for permission checking from AArch32 stage 1 translations

FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level,
                                     bits(4) domain, bit NS, AccType acctype, boolean iswrite)
assert ELUsingAArch32(S1TranslationRegime());

if PSTATE.EL != EL2 then
    wxn = SCTL.R.WXN == '1';
    if TTBCR.EAE == '1' || SCTL.R.AFE == '1' || perms.ap<0> == '1' then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';
    else
        priv_r = perms.ap<2:1> != '00';
        priv_w = perms.ap<2:1> == '01';
        user_r = perms.ap<1> == '1';
        user_w = FALSE;
    uwxn = SCTL.R.UWXN == '1';

    ispriv = AArch32.AccessIsPrivileged(acctype);

    pan = if HavePANExt() then PSTATE.PAN else '0';
    is_ldst = !(acctype IN { if (pan == '1' && user_r && ispriv &&
        !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
    is_atslxp = (acctype == ) ++
        (acctype == AccType_AT && AArch32.ExecutingATSlxPInstr());
    if pan == '1' && user_r && ispriv && (is_ldst || is_atslxp) then
    )) then
        priv_r = FALSE;
        priv_w = FALSE;

    user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
    priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
        (priv_w && wxn) || (user_w && uwxn));

    if ispriv then
        (r, w, xn) = (priv_r, priv_w, priv_xn);
    else
        (r, w, xn) = (user_r, user_w, user_xn);
else
    // Access from EL2
    wxn = HSCTL.R.WXN == '1';
    r = TRUE;
    w = perms.ap<2> == '0';
    xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' then
        secure_instr_fetch = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;
        if secure_instr_fetch == '1' then xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, } then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite && ! ( ) && PSTATE.EL == EL1 && (acctype != AccType_DC AccType_ORDEREDATOMICRW IsSecure
        fail = !r || !w;
        failedread = !r;
    ) then
        fail = !w;
        failedread = FALSE;
    elsif acctype == AccType_DC then
        // DC maintenance instructions operating by VA, cannot fault from stage 1 translation.
        fail = FALSE;
    elsif iswrite then

```

```

    fail = !w;
    failedread = FALSE;
else
    fail = !r;
    failedread = TRUE;

if fail then
    secondstage = FALSE;
    s2fslwalk = FALSE;
    ipaddress = bits(40) UNKNOWN;
    return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                   !failedread, secondstage, s2fslwalk);
else
    return AArch32.NoFault();

```

## Library pseudocode for aarch32/translation/checks/AArch32.CheckS2Permission

```
// AArch32.CheckS2Permission()
// =====
// Function used for permission checking from AArch32 stage 2 translations

FaultRecord AArch32.CheckS2Permission(Permissions perms, bits(32) vaddress, bits(40) ipaddress,
                                       integer level, AccType acctype, boolean iswrite,
                                       boolean s2fslwalk)

    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    if HaveExtendedExecuteNeverExt() then
        case perms.xn:perms.xxn of
            when '00'   xn = !r;
            when '01'   xn = !r || PSTATE.EL == EL1;
            when '10'   xn = TRUE;
            when '11'   xn = !r || PSTATE.EL == EL0;
        else
            xn = !r || perms.xn == '1';
    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fslwalk then
        fail = xn;
        failedread = TRUE;
    elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, }) && !s2fslwalk then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite && !s2fslwalk then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = !iswrite;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AccType_ORDEREDATOMICRW }) && !s2fslwalk then
            fail = !r || !w;
            failedread = !r;
    elsif acctype == AccType_DC && !s2fslwalk then
        // DC maintenance instructions operating by VA, do not generate Permission faults
        // from stage 2 translation, other than from stage 1 translation table walk.
        fail = FALSE;
    elsif iswrite && !s2fslwalk then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = !iswrite;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                       !failedread, secondstage, s2fslwalk);

    else
        return AArch32.NoFault();
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

```
// AArch32.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert size IN {2,4};

    match = FALSE;
    mismatch = FALSE;

    for i = 0 to UInt\(DBGDIDR.BRPs\)
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt\_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) && DBGDSCRext.MDBGGen == '1' && AArch32.GenerateDebugExceptions() then
        acctype = AccType\_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException\_Breakpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

```
// AArch32.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch32.NoFault();

    d_side = (acctype != AccType\_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool\(Unpredictable\_BPVECTORCATCHPRI\);

    if !d_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(vaddress, size);

    if fault.type == Fault\_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch32.CheckBreakpoint(vaddress, size);

    if fault.type == Fault\_None && !d_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(vaddress, size);

    return fault;
```



## Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// Vector Catch can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);

    match = AArch32.VCRMatch(vaddress);
    if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHHALF);

    if match && DBGDSCRext.MDBGGen == '1' && AArch32.GenerateDebugExceptions() then
        acctype = AccType\_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException\_VectorCatch;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
                                     boolean iswrite, integer size)
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);

    match = FALSE;
    ispriv = AArch32.AccessIsPrivileged(acctype);

    for i = 0 to UInt(DBGDIDR.WRPs)
        match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt\_Watchpoint;
        Halt(reason);
    elsif match && DBGDSCRext.MDBGGen == '1' && AArch32.GenerateDebugExceptions() then
        debugmoe = DebugException\_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

## Library pseudocode for aarch32/translation/faults/AArch32.AccessFlagFault

```
// AArch32.AccessFlagFault()
// =====

FaultRecord AArch32.AccessFlagFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fslwalk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault\_AccessFlag, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch32/translation/faults/AArch32.AddressSizeFault

```
// AArch32.AddressSizeFault()
// =====

FaultRecord AArch32.AddressSizeFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fslwalk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault\_AddressSize, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch32/translation/faults/AArch32.AlignmentFault

```
// AArch32.AlignmentFault()
// =====

FaultRecord AArch32.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fslwalk = boolean UNKNOWN;

    return AArch32.CreateFaultRecord(Fault\_Alignment, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch32/translation/faults/AArch32.AsynchExternalAbort

```
// AArch32.AsynchExternalAbort()
// =====
// Wrapper function for asynchronous external aborts

FaultRecord AArch32.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

    type = if parity then Fault\_AsyncParity else Fault\_AsyncExternal;
    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType\_NORMAL;
    iswrite = boolean UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    secondstage = FALSE;
    s2fslwalk = FALSE;

    return AArch32.CreateFaultRecord(type, ipaddress, domain, level, acctype, iswrite, extflag,
                                     debugmoe, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch32/translation/faults/AArch32.DebugFault

```
// AArch32.DebugFault()
// =====

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fslwalk = FALSE;

    return AArch32.CreateFaultRecord(Fault\_Debug, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch32/translation/faults/AArch32.DomainFault

```
// AArch32.DomainFault()
// =====

FaultRecord AArch32.DomainFault(bits(4) domain, integer level, AccType acctype, boolean iswrite)

    ipaddress = bits(40) UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fslwalk = FALSE;

    return AArch32.CreateFaultRecord(Fault\_Domain, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch32/translation/faults/AArch32.NoFault

```
// AArch32.NoFault()
// =====

FaultRecord AArch32.NoFault()

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType\_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fslwalk = FALSE;

    return AArch32.CreateFaultRecord(Fault\_None, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch32/translation/faults/AArch32.PermissionFault

```
// AArch32.PermissionFault()
// =====

FaultRecord AArch32.PermissionFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fslwalk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord\(Fault\_Permission, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch32/translation/faults/AArch32.TranslationFault

```
// AArch32.TranslationFault()
// =====

FaultRecord AArch32.TranslationFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fslwalk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord\(Fault\_Translation, ipaddress, domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch32/translation/translation/AArch32.FirstStageTranslate

```
// AArch32.FirstStageTranslate()
// =====
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.FirstStageTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                             boolean wasaligned, integer size)

    if PSTATE.EL == EL2 then
        s1_enabled = HSCTLR.M == '1';
    elseif EL2Enabled() then
        tge = (if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE);
        dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
        s1_enabled = tge == '0' && dc == '0' && SCTLR.M == '1';
    else
        s1_enabled = SCTLR.M == '1';

    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fslwalk = FALSE;

    if s1_enabled then // First stage enabled
        use_long_descriptor_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
        if use_long_descriptor_format then
            S1 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                                s2fslwalk, size);
            permissioncheck = TRUE; domaincheck = FALSE;
        else
            S1 = AArch32.TranslationTableWalkSD(vaddress, acctype, iswrite, size);
            permissioncheck = TRUE; domaincheck = TRUE;
    else
        S1 = AArch32.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE; domaincheck = FALSE;
        if UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() then
            if S1.addrdesc.memattrs.type == MemType\_Device && S1.addrdesc.memattrs.device != DeviceType
                nTLSMD = if S1TranslationRegime() == EL2 then HSCTLR.nTLSMD else SCTLR.nTLSMD;
                if nTLSMD == '0' then
                    S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

// Check for unaligned data accesses to Device memory
if ((!wasaligned && acctype != AccType\_IFETCH) || (acctype == AccType\_DCZVA)
    && S1.addrdesc.memattrs.type == MemType\_Device && !IsFault(S1.addrdesc)) then
    S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
if !IsFault(S1.addrdesc) && domaincheck then
    (permissioncheck, abort) = AArch32.CheckDomain(S1.domain, vaddress, S1.level, acctype,
                                                iswrite);
    S1.addrdesc.fault = abort;

if !IsFault(S1.addrdesc) && permissioncheck then
    S1.addrdesc.fault = AArch32.CheckPermission(S1.perms, vaddress, S1.level,
                                                S1.domain, S1.addrdesc.paddress.NS,
                                                acctype, iswrite);

// Check for instruction fetches from Device memory not marked as execute-never. If there has
// not been a Permission Fault then the memory is not marked execute-never.
if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType\_Device &&
    acctype == AccType\_IFETCH) then
    S1.addrdesc = AArch32.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                                S1.domain, acctype, iswrite,
                                                secondstage, s2fslwalk);

return S1.addrdesc;
```

## Library pseudocode for aarch32/translation/translation/AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =====
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch32.FullTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                         boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch32.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(S1) && !(HaveNV2Ext() && acctype == AccType\_NV2REGISTER) && HasS2Translation() then
        s2fslwalk = FALSE;
        result = AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fslwalk,
                                                size);
    else
        result = S1;

    return result;
```

## Library pseudocode for aarch32/translation/translation/AArch32.SecondStageTranslate

```
// AArch32.SecondStageTranslate()
// =====
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.SecondStageTranslate(AddressDescriptor S1, bits(32) vaddress,
AccType acctype, boolean iswrite, boolean wasaligned,
boolean s2fslwalk, integer size)

assert HasS2Translation();
assert IsZero(S1.paddress.address<47:40>);
hwupdatewalk = FALSE;
if !ELUsingAArch32(EL2) then
    return AArch64.SecondStageTranslate(S1, ZeroExtend(vaddress, 64), acctype, iswrite,
wasaligned, s2fslwalk, size, hwupdatewalk);

s2_enabled = HCR.VM == '1' || HCR.DC == '1';
secondstage = TRUE;

if s2_enabled then // Second stage enabled
    ipaddress = S1.paddress.address<39:0>;
    S2 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
s2fslwalk, size);

    // Check for unaligned data accesses to Device memory
    if ((!wasaligned && acctype != AccType\_IFETCH) || (acctype == AccType\_DCZVA))
        && S2.addrdesc.memattrs.type == MemType\_Device && !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

    // Check for permissions on Stage2 translations
    if !IsFault(S2.addrdesc) then
        S2.addrdesc.fault = AArch32.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
acctype, iswrite, s2fslwalk);

    // Check for instruction fetches from Device memory not marked as execute-never. As there
    // has not been a Permission Fault then the memory is not marked execute-never.
    if (!s2fslwalk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType\_Device &&
acctype == AccType\_IFETCH) then
        domain = bits(4) UNKNOWN;
        S2.addrdesc = AArch32.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
domain, acctype, iswrite,
secondstage, s2fslwalk);

    // Check for protected table walk
    if (s2fslwalk && !IsFault(S2.addrdesc) && HCR.PTW == '1' &&
S2.addrdesc.memattrs.type == MemType\_Device) then
        domain = bits(4) UNKNOWN;
        S2.addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, S2.level, acctype,
iswrite, secondstage, s2fslwalk);

    result = CombineS1S2Desc(S1, S2.addrdesc);
else
    result = S1;

return result;
```

## Library pseudocode for aarch32/translation/translation/AArch32.SecondStageWalk

```
// AArch32.SecondStageWalk()
// =====
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch32.SecondStageWalk(AddressDescriptor S1, bits(32) vaddress, AccType acctype,
                                           boolean iswrite, integer size)

    assert HasS2Translation();

    s2fslwalk = TRUE;
    wasaligned = TRUE;
    return AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fslwalk,
                                       size);
```

## Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)

    if !ELUsingAArch32(S1TranslationRegime()) then
        return AArch64.TranslateAddress(ZeroExtend(vaddress, 64), acctype, iswrite, wasaligned,
                                       size);
    result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType\_PTW, AccType\_IC, AccType\_AT}) && !IsFault(result) then
        result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
```





```

// AArch32.TranslationTableWalkLD()
// =====
// Returns a result of a translation table walk using the Long-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkLD(bits(40) ipaddress, bits(32) vaddress,
                                         AccType acctype, boolean iswrite, boolean secondstage,
                                         boolean s2fslwalk, integer size)

if !secondstage then
    assert ELUsingAArch32(S1TranslationRegime());
else
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(40) inputaddr;          // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2

    domain = bits(4) UNKNOWN;

    descaddr.memattrs.type = MemType_Normal;

    // Fixed parameters for the page table walk:
    // grainsize = Log2(Size of Table)          - Size of Table is 4KB in AArch32
    // stride = Log2(Address per Level)         - Bits of address consumed at each level
    constant integer grainsize = 12;           // Log2(4KB page size)
    constant integer stride = grainsize - 3;    // Log2(page size / 8 bytes)

    // Derived parameters for the page table walk:
    // inputsize = Log2(Size of Input Address) - Input Address size in bits
    // level = Level to start walk from
    // This means that the number of levels after start level = 3-level

    if !secondstage then
        // First stage translation
        inputaddr = ZeroExtend(vaddress);
        el = PSTATE.EL;
        if el == AArch32.AccessUsesEL(acctype);
        if el == EL2 then
            inputsize = 32 - UInt(HTCR.T0SZ);
            basefound = inputsize == 32 || IsZero(inputaddr<31:inputsiz>);
            disabled = FALSE;
            baseregister = HTTBR;
            descaddr.memattrs = WalkAttrDecode(HTCR.SH0, HTCR.ORGNO, HTCR.IRGNO, secondstage);
            reversedescriptors = HSCTLR.EE == '1';
            lookupsecure = FALSE;
            singlepriv = TRUE;
            hierattrsdissabled = AArch32.HaveHPDEExt() && HTCR.HPD == '1';
        else
            basefound = FALSE;
            disabled = FALSE;
            t0size = UInt(TTBCR.T0SZ);
            if t0size == 0 || IsZero(inputaddr<31:(32-t0size)>) then
                inputsize = 32 - t0size;
                basefound = TRUE;
                disabled = TTBCR.EPD0 == '1';
                baseregister = TTBR0;
                descaddr.memattrs = WalkAttrDecode(TTBCR.SH0, TTBCR.ORGNO, TTBCR.IRGNO, secondstage);
                hierattrsdissabled = AArch32.HaveHPDEExt() && TTBCR.T2E == '1' && TTBCR2.HPD0 == '1';
            t1size = UInt(TTBCR.T1SZ);
            if (t1size == 0 && !basefound) || (t1size > 0 && IsOnes(inputaddr<31:(32-t1size)>)) then
                inputsize = 32 - t1size;
                basefound = TRUE;
                disabled = TTBCR.EPD1 == '1';
                baseregister = TTBR1;
                descaddr.memattrs = WalkAttrDecode(TTBCR.SH1, TTBCR.ORGNO, TTBCR.IRGNO, secondstage);
                hierattrsdissabled = AArch32.HaveHPDEExt() && TTBCR.T2E == '1' && TTBCR2.HPD1 == '1';

```

```

        reversedescriptors = SCTL.R.EE == '1';
        lookupsecure = IsSecure();
        singlepriv = FALSE;
    // The starting level is the number of strides needed to consume the input address
    level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));

else
    // Second stage translation
    inputaddr = ipaddress;
    inputsize = 32 - SInt(VTCR.T0SZ);
    // VTCR.S must match VTCR.T0SZ[3]
    if VTCR.S != VTCR.T0SZ<3> then
        (-, inputsize) = ConstrainUnpredictableInteger(32-7, 32+8, Unpredictable\_RESVTCRS);
    basefound = inputsize == 40 || IsZero(inputaddr<39:inputsize>);
    disabled = FALSE;
    descaddr.memattrs = WalkAttrDecode(VTCR.IRGN0, VTCR.ORGNO, VTCR.SH0, secondstage);
    reversedescriptors = HSCTL.R.EE == '1';
    singlepriv = TRUE;

    lookupsecure = FALSE;
    baseregister = VTTBR;
    startlevel = UInt(VTCR.SL0);
    level = 2 - startlevel;
    if level <= 0 then basefound = FALSE;

    // Number of entries in the starting level table =
    //      (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    startsizecheck = inputsize - ((3 - level)*stride + grainsize); // Log2(Num of entries)

    // Check for starting level table with fewer than 2 entries or longer than 16 pages.
    // Lower bound check is: startsizecheck < Log2(2 entries)
    // That is, VTCR.SL0 == '00' and SInt(VTCR.T0SZ) > 1, Size of Input Address < 2^31 bytes
    // Upper bound check is: startsizecheck > Log2(pagesize/8*16)
    // That is, VTCR.SL0 == '01' and SInt(VTCR.T0SZ) < -2, Size of Input Address > 2^34 bytes
    if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

if !basefound || disabled then
    level = 1; // AArch64 reports this as a level 0 fault
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                    secondstage, s2fslwalk);

    return result;

if !IsZero(baseregister<47:40>) then
    level = 0;
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite,
                                                    secondstage, s2fslwalk);

    return result;

// Bottom bound of the Base address is:
//      Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
//      (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize); // Log2(Num of entries*8)
baseaddress = baseregister<39:baselowerbound>:Zeros(baselowerbound);

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

addrselecttop = inputsize - 1;

repeat
    addrselectbottom = (3-level)*stride + grainsize;

    bits(40) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
    descaddr.paddress.address = ZeroExtend(baseaddress OR index);
    descaddr.paddress.NS = ns_table;

    // If there are two stages of translation, then the first stage table walk addresses

```

```

// are themselves subject to translation
if secondstage || !HasS2Translation() || (HaveNV2Ext() && acctype == AccType_NV2REGISTER) then
    descaddr2 = descaddr;
else
    descaddr2 = AArch32.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
    // Check for a fault on the stage 2 walk
    if IsFault(descaddr2) then
        result.addrdesc.fault = descaddr2.fault;
        return result;

// Update virtual address for abort functions
descaddr2.vaddress = ZeroExtend(vaddress);

accdesc = CreateAccessDescriptorPTW(vaddress);

accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fslwalk, level);
(acctype, secondstage, s2fslwalk, level);
desc = _Mem[descaddr2, 8, accdesc];

if reversedescriptors then desc = BigEndianReverse(desc);

if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
    // Fault (00), Reserved (10), or Block (01) at level 3.
    // Fault (00), Reserved (10), or Block (01) at level 3
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
        iswrite, secondstage, s2fslwalk);
    return result;

// Valid Block, Page, or Table entry
if desc<1:0> == '01' || level == 3 then // Block (01) or Page (11)
    blocktranslate = TRUE;
else // Table (11)
    if !IsZero(desc<47:40>) then
        result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype,
            iswrite, secondstage, s2fslwalk);
        return result;

baseaddress = desc<39:grainsize>:Zeros(grainsize);
if !secondstage then
    // Unpack the upper and lower table attributes
    ns_table = ns_table OR desc<63>;
if !secondstage && !hierattrsdissabled then
    ap_table<1> = ap_table<1> OR desc<62>; // read-only

    xn_table = xn_table OR desc<60>;
    // pxn_table and ap_table[0] apply only in EL1&0 translation regimes
    if !singlepriv then
        pxn_table = pxn_table OR desc<59>;
        ap_table<0> = ap_table<0> OR desc<61>; // privileged

    level = level + 1;
    addrselecttop = addrselectbottom - 1;
    blocktranslate = FALSE;
until blocktranslate;

// Check the output address is inside the supported range
if !IsZero(desc<47:40>) then
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype,
        iswrite, secondstage, s2fslwalk);
    return result;

// Unpack the descriptor into address and upper and lower block attributes
outputaddress = desc<39:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
// Check the access flag
if desc<10> == '0' then
    result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
        iswrite, secondstage, s2fslwalk);
    return result;

xn = desc<54>; // Bit[54] of the block/page descriptor h
pxn = desc<53>; // Bit[53] of the block/page descriptor h

```

```

ap = desc<7:6>:'1'; // Bits[7:6] of the block/page descriptor
contiguousbit = desc<52>;
nG = desc<11>;
sh = desc<9:8>;
memattr = desc<5:2>; // AttrIndx and NS bit in stage 1

result.domain = bits(4) UNKNOWN; // Domains not used
result.level = level;
result.blocksize = 2^((3-level)*stride + grainsize);

// Stage 1 translation regimes also inherit attributes from the tables
if !secondstage then
    result.perms.xn = xn OR xn_table;
    result.perms.ap<2> = ap<2> OR ap_table<1>; // Force read-only
    // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
    if !singlepriv then
        result.perms.ap<1> = ap<1> AND NOT(ap_table<0>); // Force privileged only
        result.perms.pxn = pxn OR pxn_table;
        // Pages from Non-secure tables are marked non-global in Secure EL1&0
        if IsSecure() then
            result.nG = nG OR ns_table;
        else
            result.nG = nG;
    else
        result.perms.ap<1> = '1';
        result.perms.pxn = '0';
        result.nG = '0';
    result.GP = desc<50>; // Stage 1 block or pages might be guarded
    result.perms.ap<0> = '1';
    result.addrdesc.memattr = AArch32.S1AttrDecode(sh, memattr<2:0>, acctype);
    result.addrdesc.paddress.NS = memattr<3> OR ns_table;
else
    result.perms.ap<2:1> = ap<2:1>;
    result.perms.ap<0> = '1';
    result.perms.xn = xn;
    if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
    result.perms.pxn = '0';
    result.nG = '0';
    if s2fslwalk then
        result.addrdesc.memattr = S2AttrDecode(sh, memattr, AccType\_PTW);
    else
        result.addrdesc.memattr = S2AttrDecode(sh, memattr, acctype);
    result.addrdesc.paddress.NS = '1';

result.addrdesc.paddress.address = ZeroExtend(outputaddress);
result.addrdesc.fault = AArch32.NoFault();
result.contiguous = contiguousbit == '1';
if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;

return result;

```



```

// AArch32.TranslationTableWalkSD()
// =====
// Returns a result of a translation table walk using the Short-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkSD(bits(32) vaddress, AccType acctype, boolean iswrite,
                                         integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    // This is only called when address translation is enabled
    TLBRecord result;
    AddressDescriptor l1descaddr;
    AddressDescriptor l2descaddr;
    bits(40) outputaddress;

    // Variables for Abort functions
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fslwalk = FALSE;
    NS = bit UNKNOWN;

    // Default setting of the domain
    domain = bits(4) UNKNOWN;

    // Determine correct Translation Table Base Register to use.
    bits(64) ttbr;
    n = UInt(TTBCR.N);
    if n == 0 || IsZero(vaddress<31:(32-n)>) then
        ttbr = TTBR0;
        disabled = (TTBCR.PD0 == '1');
    else
        ttbr = TTBR1;
        disabled = (TTBCR.PD1 == '1');
        n = 0; // TTBR1 translation always works like N=0 TTBR0 translation

    // Check this Translation Table Base Register is not disabled.
    if disabled then
        level = 1;
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                         secondstage, s2fslwalk);

        return result;

    // Obtain descriptor from initial lookup.
    l1descaddr.paddress.address = ZeroExtend(ttbr<31:14-n>:vaddress<31-n:20>:'00');
    l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
    IRGN = ttbr<0>:ttbr<6>; // TTBR.IRGN
    RGN = ttbr<4:3>; // TTBR.RGN
    SH = ttbr<1>:ttbr<5>; // TTBR.S:TTBR.NOS
    l1descaddr.memattrs = WalkAttrDecode(SH, RGN, IRGN, secondstage);

    if !HaveEL(EL2) || (IsSecure() && !IsSecureEL2Enabled()) then
        // if only 1 stage of translation
        l1descaddr2 = l1descaddr;
    else
        l1descaddr2 = AArch32.SecondStageWalk(l1descaddr, vaddress, acctype, iswrite, 4);
        // Check for a fault on the stage 2 walk
        if IsFault(l1descaddr2) then
            result.addrdesc.fault = l1descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    l1descaddr2.vaddress = ZeroExtend(vaddress);

    accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fslwalk, level);
    l1desc = \_Mem[l1descaddr2, 4, accdesc];

    if SCTL.R.EE == '1' then l1desc = BigEndianReverse(l1desc);

```

```

// Process descriptor from initial lookup.
case l1desc<1:0> of
  when '00' // Fault, Reserved
    level = 1;
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                    iswrite, secondstage, s2fslwalk);
    return result;

  when '01' // Large page or Small page
    domain = l1desc<8:5>;
    level = 2;
    pxn = l1desc<2>;
    NS = l1desc<3>;

    // Obtain descriptor from level 2 lookup.
    l2descaddr.paddress.address = ZeroExtend(l1desc<31:10>:vaddress<19:12>:'00');
    l2descaddr.paddress.NS = if IsSecure() then '0' else '1';
    l2descaddr.memattrs = l1descaddr.memattrs;

    if !HaveEL\(EL2\) || (IsSecure() && !IsSecureEL2Enabled()) then
      // if only 1 stage of translation
      l2descaddr2 = l2descaddr;
    else
      l2descaddr2 = AArch32.SecondStageWalk(l2descaddr, vaddress, acctype, iswrite, 4);
      // Check for a fault on the stage 2 walk
      if IsFault(l2descaddr2) then
        result.addrdesc.fault = l2descaddr2.fault;
        return result;

    // Update virtual address for abort functions
    l2descaddr2.vaddress = ZeroExtend(vaddress);

    accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fslwalk, level);
    l2desc = _Mem[l2descaddr2, 4, accdesc];

    if SCTLRR.EE == '1' then l2desc = BigEndianReverse(l2desc);

    // Process descriptor from level 2 lookup.
    if l2desc<1:0> == '00' then
      result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                    iswrite, secondstage, s2fslwalk);
      return result;

    nG = l2desc<11>;
    S = l2desc<10>;
    ap = l2desc<9,5:4>;

    if SCTLRR.AFE == '1' && l2desc<4> == '0' then
      // ARMv8 VMSAv8-32 does not support hardware management of the Access flag.
      result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                    iswrite, secondstage, s2fslwalk);
      return result;

    if l2desc<1> == '0' then // Large page
      xn = l2desc<15>;
      tex = l2desc<14:12>;
      c = l2desc<3>;
      b = l2desc<2>;
      blocksize = 64;
      outputaddress = ZeroExtend(l2desc<31:16>:vaddress<15:0>);
    else // Small page
      tex = l2desc<8:6>;
      c = l2desc<3>;
      b = l2desc<2>;
      xn = l2desc<0>;
      blocksize = 4;
      outputaddress = ZeroExtend(l2desc<31:12>:vaddress<11:0>);

  when '1x' // Section or Supersection

```



```

    NS = l1desc<19>;
    nG = l1desc<17>;
    S = l1desc<16>;
    ap = l1desc<15,11:10>;
    tex = l1desc<14:12>;
    xn = l1desc<4>;
    c = l1desc<3>;
    b = l1desc<2>;
    pxn = l1desc<0>;
    level = 1;

    if SCTL.R.AFE == '1' && l1desc<10> == '0' then
        // ARMv8 VMSAv8-32 does not support hardware management of the Access flag.
        result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                             iswrite, secondstage, s2fslwalk);
        return result;

    if l1desc<18> == '0' then
        // Section
        domain = l1desc<8:5>;
        blocksize = 1024;
        outputaddress = ZeroExtend(l1desc<31:20>:vaddress<19:0>);
    else
        // Supersection
        domain = '0000';
        blocksize = 16384;
        outputaddress = l1desc<8:5>:l1desc<23:20>:l1desc<31:24>:vaddress<23:0>;

// Decode the TEX, C, B and S bits to produce the TLBRecord's memory attributes
if SCTL.R.TRE == '0' then
    if RemapRegsHaveResetValues() then
        result.addrdesc.memattrs = AArch32.DefaultTEXDecode(tex, c, b, S, acctype);
    else
        result.addrdesc.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
else
    result.addrdesc.memattrs = AArch32.RemappedTEXDecode(tex, c, b, S, acctype);

// Set the rest of the TLBRecord, try to add it to the TLB, and return it.
result.perms.ap = ap;
result.perms.xn = xn;
result.perms.pxn = pxn;
result.nG = nG;
result.domain = domain;
result.level = level;
result.blocksize = blocksize;
result.addrdesc.paddress.address = ZeroExtend(outputaddress);
result.addrdesc.paddress.NS = if IsSecure() then NS else '1';
result.addrdesc.fault = AArch32.NoFault();

return result;

```

## Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

```

boolean RemapRegsHaveResetValues();

```

## Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =====
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccType acctype, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.BRPs);

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                     linked, DBGBCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAnyAArch32() && size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);

    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);

    match = value_match && state_match && enabled;

    return match;
```



```

// AArch64.BreakpointValueMatch()
// =====

boolean AArch64.BreakpointValueMatch(integer n, bits(64) vaddress, boolean linked_to)

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n > UInt(ID_AA64DFR0_EL1.BRPs) then
    (c, n) = ConstrainUnpredictableInteger(0, UInt(ID_AA64DFR0_EL1.BRPs), Unpredictable_BPNOTIMPL);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return FALSE;

// If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
// call from StateMatch for linking).
if DBGBCR_EL1[n].E == '0' then return FALSE;

context_aware = (n >= UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
type = DBGBCR_EL1[n].BT;

if ((type IN {'011x', '11xx'} && !HaveVirtHostExt()) || // Context matching
    type == '010x' || // Reserved
    (type != '0x0x' && !context_aware) || // Context matching
    (type == '1xxx' && !HaveEL(EL2))) then // EL2 extension
    (c, type) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (type == '0x0x');
match_vmid = (type == '10xx');
match_cid = (type == '001x');
match_cid1 = (type IN {'101x', 'x11x'});
match_cid2 = (type == '11xx');
linked = (type == 'xxx1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, or if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return FALSE;

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return FALSE;

// Do the comparison.
if match_addr then
    byte = UInt(vaddress<1:0>);
    if HaveAnyAArch32() then
        // T32 instructions can be executed at EL0 in an AArch64 translation regime.
        assert byte IN {0,2}; // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
    else
        assert byte == 0; // "vaddress" is word aligned
        byte_select_match = TRUE; // DBGBCR_EL1[n].BAS<byte> is RES1
    top = AddrTop(vaddress, TRUE, PSTATE.EL);
    BVR_match = vaddress<top:2> == DBGBCR_EL1[n]<top:2> && byte_select_match;
elseif match_cid then
    if IsInHost() then
        BVR_match = (CONTEXTIDR_EL2 == DBGBCR_EL1[n]<31:0>);
    else
        BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1 == DBGBCR_EL1[n]<31:0>);
elseif match_cid1 then
    BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1 == DBGBCR_EL1[n]<31:0>);

```

```

if match_vmid then
    if !Have16bitVMID() || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGBVR_EL1[n]<47:32>;
    BXVR_match = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        !IsInHost() &&
        vmid == bvr_vmid);
elseif match_cid2 then
    BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
        DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2);

bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

return match;

```

## Library pseudocode for aarch64/debug/breakpoint/AArch64.StateMatch

```
// AArch64.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreakpnt, AccType acctype, boolean ispriv)
// "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

// If parameters are set to a reserved type, behaves as either disabled or a defined type
if ((HMC:SSC:PxC) IN {'011xx','100x0','101x0','11010','11101','1111x'}) || // Reserved
    (HMC == '0' && PxC == '00' && (!isbreakpnt || !HaveAArch32EL\(EL1\))) || // Usr/Svc/Sys
    (SSC IN {'01','10'} && !HaveEL\(EL3\)) || // No EL3
    (HMC:SSC != '000' && HMC:SSC != '111' && !HaveEL\(EL3\) && !HaveEL\(EL2\)) || // No EL3/EL2
    (HMC:SSC:PxC == '11100' && !HaveEL\(EL2\))) then // No EL2
    (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits\(Unpredictable\_RESBPWPCTRL\);
    assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
    if c == Constraint\_DISABLED then return FALSE;
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

EL3_match = HaveEL\(EL3\) && HMC == '1' && SSC<0> == '0';
EL2_match = HaveEL\(EL2\) && HMC == '1';
EL1_match = PxC<0> == '1';
EL0_match = PxC<1> == '1';

el = if HaveNV2Ext\(\) && acctype == AccType\_NV2REGISTER then EL2 else PSTATE.EL;
if !ispriv && !isbreakpnt then
    priv_match = EL0_match;
else
    case el of
        when EL3 priv_match = EL3_match;
        when EL2 priv_match = EL2_match;
        when EL1 priv_match = EL1_match;
        when EL0 priv_match = EL0_match;

case SSC of
    when '00' security_state_match = TRUE; // Both
    when '01' security_state_match = !IsSecure\(\); // Non-secure only
    when '10' security_state_match = IsSecure\(\); // Secure only
    when '11' security_state_match = TRUE; // Both

if linked then
    // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
    // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    lbn = UInt\(LBN\);
    first_ctx_cmp = (UInt\(ID\_AA64DFR0\_EL1.BRPs\) - UInt\(ID\_AA64DFR0\_EL1.CTX\_CMPs\));
    last_ctx_cmp = UInt\(ID\_AA64DFR0\_EL1.BRPs\);
    if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
        (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable\_BPNOTCT);
        assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNKNOWN};
        case c of
            when Constraint\_DISABLED return FALSE; // Disabled
            when Constraint\_NONE linked = FALSE; // No linking
            // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

if linked then
    vaddress = bits(64) UNKNOWN;
    linked_to = TRUE;
    linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

return priv_match && security_state_match && (!linked || linked_match);
```

## Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =====

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);
```

## Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = HaveEL(EL2) && !secure && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');
    target = (if route_to_el2 then EL2 else EL1);

    enabled = !HaveEL(EL3) || !secure || MDCR_EL3.SDD == '0';

    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from);

    return enabled;
```

## Library pseudocode for aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```
// AArch64.CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch64.CheckForPMUOverflow()

    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSSET_EL0<31> == '1';
    for n = 0 to UInt(PMCR_EL0.N) - 1
        if HaveEL(EL2) then
            E = (if n < UInt(MDCR_EL2.HPMN) then PMCR_EL0.E else MDCR_EL2.HPME);
        else
            E = PMCR_EL0.E;
        if E == '1' && PMINTENSET_EL1<n> == '1' && PMOVSSET_EL0<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID\_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn\_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCCLR_EL0.)

    return pmuirq;
```

## Library pseudocode for aarch64/debug/pmu/AArch64.CountEvents

```
// AArch64.CountEvents()
// =====
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch64.CountEvents(integer n)
    assert n == 31 || n < UInt(PMCR_EL0.N);

    // Event counting is disabled in Debug state
    debug = Halted();

    // In Non-secure state, some counters are reserved for EL2
    if HaveEL(EL2) then
        E = if n < UInt(MDCR_EL2.HPMN) || n == 31 then PMCR_EL0.E else MDCR_EL2.HPME;
    else
        E = PMCR_EL0.E;
    enabled = E == '1' && PMCNTENSET_EL0<n> == '1';

    if !IsSecure() then
        // Event counting in Non-secure state is allowed unless all of:
        // * EL2 and the HPMD Extension are implemented
        // * Executing at EL2
        // * PMNx is not reserved for EL2
        // * MDCR_EL2.HPMD == 1
        if HaveHPMDExt() && PSTATE.EL == EL2 && (n < UInt(MDCR_EL2.HPMN) || n == 31) then
            prohibited = (MDCR_EL2.HPMD == '1');
        else
            prohibited = FALSE;
    else
        // Event counting in Secure state is prohibited unless any one of:
        // * EL3 is not implemented
        // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
        prohibited = HaveEL(EL3) && MDCR_EL3.SPME == '0';

    // The IMPLEMENTATION DEFINED authentication interface might override software controls
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // For the cycle counter, PMCR_EL0.DP enables counting when otherwise prohibited
    if prohibited && n == 31 then prohibited = (PMCR_EL0.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M} bits
    filter = if n == 31 then PMCCFILTR else PMEVTYPER[n];

    P = filter<31>;
    U = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';
    M = if HaveEL(EL3) then filter<26> else '0';

    case PSTATE.EL of
        when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
        when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
        when EL2 filtered = (NSH == '0');
        when EL3 filtered = (M != P);

    return !debug && enabled && !prohibited && !filtered;
```



## Library pseudocode for aarch64/debug/statisticalprofiling/CheckProfilingBufferAccess

```
// CheckProfilingBufferAccess()
// =====

SysRegAccess CheckProfilingBufferAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;

    if EL2Enabled() && PSTATE.EL == EL1 && MDCR_EL2.E2PB<0> != '1' then
        return SysRegAccess_TrapToEL2;

    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;

    return SysRegAccess_OK;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CheckStatisticalProfilingAccess

```
// CheckStatisticalProfilingAccess()
// =====

SysRegAccess CheckStatisticalProfilingAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;

    if EL2Enabled() && PSTATE.EL == EL1 && MDCR_EL2.TPMS == '1' then
        return SysRegAccess_TrapToEL2;

    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;

    return SysRegAccess_OK;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// =====

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```
// CollectContextIDR2()
// =====

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// =====

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    if !secure && HaveEL(EL2) then
        return PMSCR_EL2.PA == '1' && (el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectRecord

```
// CollectRecord()
// =====

boolean CollectRecord(bits(64) events, integer total_latency, OpType optype)
    assert StatisticalProfilingEnabled\(\);
    if PMSFCR_EL1.FE == '1' then
        e = events<63:48,31:24,15:12,7,5,3,1>;
        m = PMSEVFR_EL1<63:48,31:24,15:12,7,5,3,1>;
        // Check for UNPREDICTABLE case
        if IsZero(PMSEVFR_EL1) && ConstrainUnpredictableBool(Unpredictable\_ZEROPMSEVFR) then return FALSE;
        if !IsZero(NOT(e) AND m) then return FALSE;
    if PMSFCR_EL1.FT == '1' then
        // Check for UNPREDICTABLE case
        if IsZero(PMSFCR_EL1.<B,LD,ST>) && ConstrainUnpredictableBool(Unpredictable\_NOOPTYPES) then
            return FALSE;
        case optype of
            when OpType\_Branch if PMSFCR_EL1.B == '0' then return FALSE;
            when OpType\_Load if PMSFCR_EL1.LD == '0' then return FALSE;
            when OpType\_Store if PMSFCR_EL1.ST == '0' then return FALSE;
            when OpType\_LoadAtomic if PMSFCR_EL1.<LD,ST> == '00' then return FALSE;
            otherwise return FALSE;
    if PMSFCR_EL1.FL == '1' then
        if IsZero(PMSLATFR_EL1.MINLAT) && ConstrainUnpredictableBool(Unpredictable\_ZEROMINLATENCY) then
            return FALSE;
        if total_latency < UInt(PMSLATFR_EL1.MINLAT) then return FALSE;
    return TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// =====

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled\(\) then return TimeStamp\_None;
    (secure, el) = ProfilingBufferOwner();
    if el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp\_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp\_None;
    if EL2Enabled() then
        pct = PMSCR_EL2.PCT == '1' && (el == EL2 || PMSCR_EL1.PCT == '1');
    else
        pct = PMSCR_EL1.PCT == '1';
    return (if pct then TimeStamp\_Physical else TimeStamp\_Virtual);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/OpType

```
enumeration OpType {
    OpType_Load,           // Any memory-read operation other than atomics, compare-and-swap, and sw
    OpType_Store,          // Any memory-write operation, including atomics without return
    OpType_LoadAtomic,     // Atomics with return, compare-and-swap and swap
    OpType_Branch,         // Software write to the PC
    OpType_Other           // Any other class of operation
};
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// =====

boolean ProfilingBufferEnabled()
    if !HaveStatisticalProfiling() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    non_secure_bit = if secure then '0' else '1';
    return (!ELUsingAArch32(el) && non_secure_bit == SCR_EL3.NS &&
        PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// =====

(boolean, bits(2)) ProfilingBufferOwner()
    secure = if HaveEL(EL3) then (MDCR_EL3.NSPB<1> == '0') else IsSecure();
    el = if !secure && HaveEL(EL2) && MDCR_EL2.E2PB == '00' then EL2 else EL1;
    return (secure, el);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();
```

## Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =====

boolean StatisticalProfilingEnabled()
    if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    in_host = EL2Enabled() && HCR_EL2.TGE == '1';
    (secure, el) = ProfilingBufferOwner();
    if UInt(el) < UInt(PSTATE.EL) || secure != IsSecure() || (in_host && el == EL1) then
        return FALSE;

    case PSTATE.EL of
        when EL3 Unreachable();
        when EL2 spe_bit = PMSCR_EL2.E2SPE;
        when EL1 spe_bit = PMSCR_EL1.E1SPE;
        when EL0 spe_bit = (if in_host then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);

    return spe_bit == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SysRegAccess

```
enumeration SysRegAccess { SysRegAccess_OK,
                           SysRegAccess_UNDEFINED,
                           SysRegAccess_TrapToEL1,
                           SysRegAccess_TrapToEL2,
                           SysRegAccess_TrapToEL3 };
```

## Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```
enumeration TimeStamp { TimeStamp_None,
                        TimeStamp_Virtual,
                        TimeStamp_Physical };
```

## Library pseudocode for aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// =====
// Take an exception in Debug state to an Exception Level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    sync_errors = (PSTATE.EL); HaveIESB() && SCTLRL[0].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLRL[0].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(target_el);

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el; PSTATE.nRW = '0'; PSTATE.SP = '1';

    SPSPR[] = bits(32) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000'; PSTATE.T = '0'; // PSTATE.J is RES0
    if HaveSSBSEExt() then PSTATE.SSBS = bits(1) UNKNOWN;
    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    if HaveBTIExt() then
        if exception.type == Exception_Breakpoint then
            DSPSR_EL0<11:10> = PSTATE.BTYPE;
        else
            DSPSR_EL0<11:10> = if ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE) then '00' else PS
            PSTATE.BTYPE = '00';

    PSTATE.IL = '0';
    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000'; PSTATE.T = '0'; // PSTATE.J is RES0
    if HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) && SCTLRL[0].SPAN == '1'
        PSTATE.PAN = '1';

    if
        EDSCR.ERR = '1'; HaveMTEExtUpdateEDSCRFields() then PSTATE.TCO = '1';
    ();// // Update EDSCR processor state flags.

    EDSCR.ERR = '1'; // SCTLRL[0].IESB might be ignored in Debug state.
    if
        ( ) && SCTLRL[0].IESB == '1' && ConstrainUnpredictableBool(Unpredictable_IESBinDebug) UpdateEDSCRFieldsH

    if sync_errors then then
        SynchronizeErrors();

    EndOfInstruction();
```

## Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// =====

boolean AArch64.WatchpointByteMatch(integer n, AccType acctype, bits(64) vaddress)

    el = if HaveNV2Ext() && acctype == AccType\_NV2REGISTER then EL2 else PSTATE.EL;
    top = AddrTop(vaddress, FALSE, el);
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;           // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable\_WPMASKANDBAS);
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1)); MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                       // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable\_WPBASCONTIGUOUS);
            bottom = 3;                                           // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable\_RESWPMASK);
        assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNKNOWN};
        case c of
            when Constraint\_DISABLED return FALSE;           // Disabled
            when Constraint\_NONE mask = 0;                   // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable\_WPMASKEDBITS);
    else
        WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

    return WVR_match && byte_select_match;
```

## Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =====
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
                                AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.WRPs);

    // "ispriv" is FALSE for LDTR/STTR instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);

    ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, acctype, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// =====
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

    if IsDebugException(fault) then
        if fault.acctype == AccType\_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException\_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(vaddress, fault);
    elseif fault.acctype == AccType\_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception type, FaultRecord fault, bits(64) vaddress)

    exception = ExceptionSyndrome(type);

    d_side = type IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint};

    exception.syndrome = AArch64.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = fault.ipaddress.NS;
        exception.ipaddress = fault.ipaddress.address;
    else
        exception.ipavalid = FALSE;

    return exception;
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// =====

AArch64.CheckPCAlignment()

    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// =====

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} && (HCR_EL2.TGE == '1' ||
        (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
        (HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER) ||
        IsSecondStage(fault)));

    bits(64) preferred_exception_return = ThisInstrAddr();
    if (vect_offset = 0x0;
    if HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
        IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress);
    else
        exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);
    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// =====

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
    // External aborts on instruction fetch must be taken synchronously
    if route_to_el3 = HaveDoubleFaultExt() then assert fault.type != Fault AsyncExternal;
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
        (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault))));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception InstructionAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// =====
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception PCAlignment);
    exception.vaddress = ThisInstrAddr();

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// =====
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception SPAlignment);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```



## Library pseudocode for aarch64/exceptions/aborts/BranchTargetException

```
// BranchTargetException
// =====
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)

    route_to_el2 = EL2Enabled() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_BranchTarget);
    exception.syndrome<1:0> = PSTATE.BTYPE;
    exception.syndrome<24:2> = Zeros(); // RES0

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/EffectiveTCF

```
// EffectiveTCF()
// =====
// Returns the TCF field applied to Tag Check Fails in the given Exception Level

bits(2) EffectiveTCF(bits(2) el)
    if el == EL3 then
        tcf = SCTLR_EL3.TCF;
    elsif el == EL2 then
        tcf = SCTLR_EL2.TCF;
    elsif el == EL1 then
        tcf = SCTLR_EL1.TCF;
    elsif el == EL0 && HCR_EL2.<E2H,TGE> == '11' then
        tcf = SCTLR_EL2.TCF0;
    elsif el == EL0 && HCR_EL2.<E2H,TGE> != '11' then
        tcf = SCTLR_EL1.TCF0;

    return tcf;
```

## Library pseudocode for aarch64/exceptions/aborts/RecordTagCheckFail

```
// RecordTagCheckFail()
// =====
// Records a tag fail exception into the appropriate TCFR_ELx

ReportTagCheckFail(bits(2) el, bit ttbr)
    if el == EL3 then
        assert ttbr == '0';
        TFSR_EL3.TF0 = '1';
    elsif el == EL2 then
        if ttbr == '0' then
            TFSR_EL2.TF0 = '1';
        else
            TFSR_EL2.TF1 = '1';
    elsif el == EL1 then
        if ttbr == '0' then
            TFSR_EL1.TF0 = '1';
        else
            TFSR_EL1.TF1 = '1';
    elsif el == EL0 then
        if ttbr == '0' then
            TFSR_EL1.TF0 = '1';
        else
            TFSR_EL1.TF1 = '1';
```

## Library pseudocode for aarch64/exceptions/aborts/TagCheckFail

```
// TagCheckFail()
// =====
// Handle a tag check fail condition

TagCheckFail(bits(64) vaddress, boolean iswrite)
    bits(2) tcf = EffectiveTCF(PSTATE.EL);
    if tcf == '01' then
        TagCheckFault(vaddress, iswrite);
    elsif tcf == '10' then
        ReportTagCheckFail(PSTATE.EL, vaddress<55>);
```

## Library pseudocode for aarch64/exceptions/aborts/TagCheckFault

```
// TagCheckFault()
// =====
// Raise a tag check fail exception.

TagCheckFault(bits(64) va, boolean write)
    bits(2) target_el;
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset = 0x0;

    if PSTATE.EL == EL0 then
        target_el = if HCR_EL2.TGE == 0 then EL1 else EL2;
    else
        target_el = PSTATE.EL;

    exception = ExceptionSyndrome(Exception DataAbort);
    exception.syndrome<5:0> = '010001';
    if write then
        exception.syndrome<6> = '1';
    exception.vaddress = va;

    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// =====

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
    exception = ExceptionSyndrome(Exception FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0,EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====

AArch64.TakePhysicalSErrorException(boolean impdef_syndrome, bits(24) syndrome)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome<24> = if impdef_syndrome then '1' else '0';
    exception.syndrome<23:0> = syndrome;

    ClearPendingPhysicalSError();

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =====

AArch64.TakeVirtualFIQException()
    assert EL2Enabled() && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1'; // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;

    exception = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =====

AArch64.TakeVirtualIRQException()
    assert EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1'; // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception\_IRQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// =====

AArch64.TakeVirtualSErrorException(boolean impdef_syndrome, bits(24) syndrome)

    assert EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1'; // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception\_SError);
    if HaveRASExt\(\) then
        exception.syndrome<24> = VESR_EL2.IDS;
        exception.syndrome<23:0> = VESR_EL2.ISS;
    else
        exception.syndrome<24> = if impdef_syndrome then '1' else '0';
        if impdef_syndrome then exception.syndrome<23:0> = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =====

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1} &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception\_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// =====

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// =====

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// =====
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =====

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0,EL1} &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception\_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// =====
// Return the Exception Class and Instruction Length fields for reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception type, bits(2) target_el)

    il = if ThisInstrLength() == 32 then '1' else '0';
    from_32 = UsingAArch32();
    assert from_32 || il == '1';           // AArch64 instructions always 32-bit

    case type of
        when Exception Uncategorized      ec = 0x00; il = '1';
        when Exception WFxTrap             ec = 0x01;
        when Exception CP15RRTTrap         ec = 0x03;          assert from_32;
        when Exception CP15RRTTrap         ec = 0x04;          assert from_32;
        when Exception CP14RRTTrap         ec = 0x05;          assert from_32;
        when Exception CP14DTTTrap         ec = 0x06;          assert from_32;
        when Exception AdvSIMDFPAccessTrap ec = 0x07;
        when Exception FPIDTrap            ec = 0x08;
        when Exception PACTrap             ec = 0x09;
        when Exception CP14RRTTrap         ec = 0x0C;          assert from_32;
        when Exception BranchTarget       ec = 0x0D;
        when Exception IllegalState       ec = 0x0E; il = '1';
        when Exception SupervisorCall     ec = 0x11;
        when Exception HypervisorCall     ec = 0x12;
        when Exception MonitorCall       ec = 0x13;
        when Exception SystemRegisterTrap ec = 0x18;          assert !from_32;
        when Exception SVEAccessTrap       ec = 0x19;          assert !from_32;
        when Exception ERetTrap            ec = 0x1A;
        when Exception InstructionAbort    ec = 0x20; il = '1';
        when Exception PCAlignment        ec = 0x22; il = '1';
        when Exception DataAbort          ec = 0x24;
        when Exception NV2DataAbort        ec = 0x25;
        when Exception SPAlignment        ec = 0x26; il = '1'; assert !from_32;
        when Exception FPTrappedException ec = 0x28;
        when Exception SError             ec = 0x2F; il = '1';
        when Exception Breakpoint         ec = 0x30; il = '1';
        when Exception SoftwareStep        ec = 0x32; il = '1';
        when Exception Watchpoint         ec = 0x34; il = '1';
        when Exception SoftwareBreakpoint ec = 0x38;
        when Exception VectorCatch       ec = 0x3A; il = '1'; assert from_32;
        otherwise                          Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    return (ec,il);
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =====
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception type = exception.type;

    (ec,il) = AArch64.ExceptionClass(type, target_el);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = ec<5:0>:il:iss;

    if type IN {Exception_InstructionAbort, Exception_PCAalignment, Exception_DataAbort,
                Exception_NV2DataAbort,
                Exception_Watchpoint} then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if target_el == EL2 then
        if exception.ipavalid then
            HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
            if HaveSecureEL2Ext() then
                if IsSecureEL2Enabled() then
                    HPFAR_EL2.NS = exception.NS;
                else
                    HPFAR_EL2.NS = '0';
            else
                HPFAR_EL2<43:4> = bits(40) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch64.ResetControlRegisters(boolean cold_reset);
```



## Library pseudocode for aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// =====
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert !HighestELUsingAArch32\(\);

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL\(EL3\) then
        PSTATE.EL = EL3;
    elsif HaveEL\(EL2\) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset the system registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';           // Select stack pointer
    PSTATE.<D,A,I,F> = '1111'; // All asynchronous exceptions masked
    PSTATE.SS = '0';           // Clear software step bit
    PSTATE.DIT = '0';          // PSTATE.DIT is reset to 0 when resetting into AArch64
    PSTATE.IL = '0';           // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;               // IMPLEMENTATION DEFINED reset vector

    if HaveEL\(EL3\) then
        rv = RVBAR_EL3;
    elsif HaveEL\(EL2\) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;
    // The reset vector must be correctly aligned
    assert IsZero(rv<63:PAMax()>) && IsZero(rv<1:0>);

    BranchTo(rv, BranchType\_RESETBranchType\_UNKNOWN);
```

## Library pseudocode for aarch64/exceptions/ieeeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// =====

AArch64.FPTrappedException(boolean is_ase, integer element, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome(Exception\_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            exception.syndrome<23> = '1'; // TFV
        else
            exception.syndrome<23> = '0'; // TFV
    else
        exception.syndrome<23> = '1'; // TFV
        exception.syndrome<10:8> = bits(3) UNKNOWN; // VECITR
        if exception.syndrome<23> == '1' then
            exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
        else
            exception.syndrome<7,4:0> = bits(6) UNKNOWN;
exception.syndrome<23> = '1'; // TFV
if is_ase then exception.syndrome<10:8> = element<2:0>; // VECITR
exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// =====
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception\_HypervisorCall);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// =====

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_MonitorCall);
    exception.syndrome<15:0> = immediate;

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// =====
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    route_to_el2 = EL2Enabled() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```



```

// AArch64.TakeException()
// =====
// Take an exception to an Exception Level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception,
                      bits(64) preferred_exception_return, integer vect_offset)
assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

sync_errors = (PSTATE.EL); HaveIESB() && SCTLR[].IESB == '1';
if HaveDoubleFaultExt() then
    sync_errors = sync_errors || (SCR\_EL3.EA == '1' && SCR\_EL3.NMEA == '1' && PSTATE.EL == EL3);
if sync_errors && InsertIESBBeforeException(target_el) then
    SynchronizeErrors();
    iesb_req = FALSE;
    sync_errors = FALSE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

SynchronizeContext();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
MaybeZeroSVEUppers(target_el);

if UInt(target_el) > UInt(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAArch32(EL2);
        else
            lower_32 = ELUsingAArch32(EL1);
    elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
        lower_32 = ELUsingAArch32(EL0);
    else
        lower_32 = ELUsingAArch32(target_el - 1);
    vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

elsif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;

spsr = GetPSRFromPSTATE();

if PSTATE.EL == if EL1 && target_el == EL1 && HaveNVEExt() && EL2Enabled() && HCR\_EL2.<NV, NV1> =
    spsr<3:2> = '10';

if HaveUAOExt() then PSTATE.UAO = '0';
if !(exception.type IN {Exception\_IRQ, Exception\_FIQ}) then
    AArch64.ReportException(exception, target_el);

PSTATE.EL = target_el; PSTATE.nRW = '0'; PSTATE.SP = '1';

if PSTATE.EL = target_el; PSTATE.nRW = '0'; PSTATE.SP = '1'; HaveBTIExt() then
    if ( exception.type IN {Exception\_SoftwareStep, Exception\_PCAalignment, Exception\_InstructionAbort,
                          Exception\_SoftwareBreakpoint, Exception\_IllegalState, Exception\_BranchTargetException}) then
        spsr_btype = PSTATE.BTYPE;
    else
        spsr_btype = if ConstrainUnpredictableBool(Unpredictable\_ZEROBTYPE) then '00' else PSTATE.BTYPE;

    spsr<11:10> = spsr_btype;

SPSR[] = spsr;
ELR[] = preferred_exception_return;

PSTATE.SS = '0';
PSTATE.<D,A,I,F> = '1111';
PSTATE.IL = '0';
if from_32 then // Coming from AArch32
    PSTATE.IT = '00000000'; PSTATE.T = '0'; // PSTATE.J is RES0
if HaveSSBSEExt() then PSTATE.SSBS = SCTLR[].DSSBS;

```

```

    if HaveBTIExt() then PSTATE.BTYPE = '00';
    PSTATE.SS = '0';
    PSTATE.<D,A,I,F> = '1111';
    PSTATE.IL = '0';
    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000'; PSTATE.T = '0'; // PSTATE.J is RES0
    if HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) && SCTLR[].SPAN == '1'
        PSTATE.PAN = '1';
    if PSTATE.PAN == '1'; HaveMTEExt() then PSTATE.TCO = '1';

    BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION);

    if HaveRASExt() && SCTLR[];

    if sync_errors then [].IESB == '1' then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalErrorInterrupts(iesb_req);

    EndOfInstruction();

```

### Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

```

// AArch64.AArch32SystemAccessTrap()
// =====
// Trapped AArch32 System register access other than due to CPTR_EL2 or CPACR_EL1.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, bits(32) aarch32_instr)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AArch32SystemAccessTrapSyndrome(aarch32_instr);

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

```

## Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrapSyndrome

```
// AArch64.AArch32SystemAccessTrapSyndrome()
// =====
// Return the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr)
    ExceptionRecord exception;
    cpnum = UInt(instr<11:8>);

    bits(20) iss = Zeros();
    if instr<27:24> == '1110' && instr<4> == '1' && instr<31:28> != '1111' then
        // MRC/MCR
        case cpnum of
            when 10    exception = ExceptionSyndrome(Exception FPIDTrap);
            when 14    exception = ExceptionSyndrome(Exception CP14RTTrap);
            when 15    exception = ExceptionSyndrome(Exception CP15RTTrap);
            otherwise Unreachable();
        iss<19:17> = instr<7:5>;        // opc2
        iss<16:14> = instr<23:21>;      // opc1
        iss<13:10> = instr<19:16>;      // CRn
        if instr<20> == '1' && instr<15:12> == '1111' then // MRC, Rt==15
            iss<9:5> = '11111';
        elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1> = instr<3:0>;        // CRm
    elsif instr<27:21> == '1100010' && instr<31:28> != '1111' then
        // MRRC/MCRR
        case cpnum of
            when 14    exception = ExceptionSyndrome(Exception CP14RRTTrap);
            when 15    exception = ExceptionSyndrome(Exception CP15RRTTrap);
            otherwise Unreachable();
        iss<19:16> = instr<7:4>;        // opc1
        if instr<19:16> == '1111' then // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

        if instr<15:12> == '1111' then // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1> = instr<3:0>;        // CRm
    elsif instr<27:25> == '110' && instr<31:28> != '1111' then
        // LDC/STC
        assert cpnum == 14;
        exception = ExceptionSyndrome(Exception CP14DTTTrap);
        iss<19:12> = instr<7:0>;        // imm8
        iss<4> = instr<23>;            // U
        iss<2:1> = instr<24,21>;        // P,W
        if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
            iss<9:5> = bits(5) UNKNOWN;
            iss<3> = '1';
        else
            iss<9:5> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rn
            iss<3> = '0';
    else
        Unreachable();
    iss<0> = instr<20>;                // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0> = iss;

    return exception;
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// =====
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled\(\) && HCR_EL2.TGE == '1');

    if route_to_el2 then
        exception = ExceptionSyndrome(Exception\_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception\_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
```





```

// AArch64.CheckAArch32SystemAccess()
// =====
// Check AArch32 System register access instruction for enables and disables

AArch64.CheckAArch32SystemAccess(bits(32) instr)
    cp_num = UInt(instr<11:8>);
    assert cp_num IN {14,15};
    // Decode the AArch32 System register access instruction
    if instr<31:28> != '1111' && instr<27:24> == '1110' && instr<4> == '1' then // MRC/MCR
        cpvt = TRUE; cpdt = FALSE; nreg = 1;
        opcl = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn = UInt(instr<19:16>);
        CRm = UInt(instr<3:0>);
    elseif instr<31:28> != '1111' && instr<27:21> == '1100010' then // MRRC/MCRR
        cpvt = TRUE; cpdt = FALSE; nreg = 2;
        opcl = UInt(instr<7:4>);
        CRm = UInt(instr<3:0>);
    elseif instr<31:28> != '1111' && instr<27:25> == '110' && instr<22> == '0' then // LDC/STC
        cpvt = FALSE; cpdt = TRUE; nreg = 0;
        opcl = 0;
        CRn = UInt(instr<15:12>);
    else
        allocated = FALSE;

//
// Coarse-grain decode into CP14 or CP15 encoding space. Each of the CPxxxInstrDecode functions
// returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
if cp_num == 14 then
    // LDC and STC only supported for c5 in CP14 encoding space
    if cpdt && CRn != 5 then
        allocated = FALSE;
    else
        // Coarse-grained decode of CP14 based on opcl field
        case opcl of
            when 0      allocated = CP14DebugInstrDecode(instr);
            when 1      allocated = CP14TraceInstrDecode(instr);
            when 7      allocated = CP14JazelleInstrDecode(instr); // JIDR only
            otherwise   allocated = FALSE; // All other values are unallocated

elseif cp_num == 15 then
    // LDC and STC not supported in CP15 encoding space
    if !cpvt then
        allocated = FALSE;
    else
        allocated = CP15InstrDecode(instr);

    // Coarse-grain traps to EL2 have a higher priority than exceptions generated because
    // the access instruction is UNDEFINED
    if AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm) then
        // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
        // User mode is UNDEFINED when the trap is disabled, then it is
        // IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
        // when the trap is enabled.
        if PSTATE.EL == EL0 && EL2Enabled() && !allocated then
            if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0" then
                UNDEFINED;
                AArch64.AArch32SystemAccessTrap(EL2, instr);

else
    allocated = FALSE;

if !allocated then
    UNDEFINED;

// If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
AArch64.CheckAArch32SystemAccessTraps(instr);

return;

```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccessEL1Traps

```
// AArch64.CheckAArch32SystemAccessEL1Traps()
// =====
// Check for configurable disables or traps to EL1 or EL2 of an AArch32 System register
// access instruction.

AArch64.CheckAArch32SystemAccessEL1Traps(bits(32) instr)
    assert PSTATE.EL == EL0;

    trap = FALSE;

    // Decode the AArch32 System register access instruction
    (op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

    if cp_num == 14 then
        if ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 == 0) || // DBGDTR
            (op == SystemAccessType\_DT && CRn == 5 && opc2 == 0)) then // DBGDTR
            trap = !Halted() && MDSCR_EL1.TDCC == '1';

        elseif opc1 == 0 then
            trap = MDSCR_EL1.TDCC == '1';

        elseif opc1 == 1 then
            trap = CPACR[][TTA] == '1';

    elseif cp_num == 15 then
        if ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0) || // PMCR
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 1) || // PMCNTE
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 2) || // PMCNTE
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 3) || // PMOVS
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 6) || // PMCEID
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 7) || // PMCEID
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 1) || // PMXEVT
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 14 && opc2 == 3) || // PMOVSS
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 14 && CRm >= 12)) then // PMEVTY
            trap = PMUSERENR_EL0.EN == '0';

        elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 14 && opc2 == 4 then // PMSW
            trap = PMUSERENR_EL0.EN == '0' && PMUSERENR_EL0.SW == '0';

        elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 0) || // PMO
            (op == SystemAccessType\_RRT && opc1 == 0 && CRm == 9)) then // PMO
            trap = PMUSERENR_EL0.EN == '0' && (write || PMUSERENR_EL0.CR == '0');

        elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 2) || // PMX
            (op == SystemAccessType\_RT && opc1 == 0 && CRn == 14 && CRm >= 8 && CRm <= 11)) then // PM
            trap = PMUSERENR_EL0.EN == '0' && (write || PMUSERENR_EL0.ER == '0');

        elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 5 then // P
            trap = PMUSERENR_EL0.EN == '0' && PMUSERENR_EL0.ER == '0';

        elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then /
            trap = CNTKCTL[][ELOPTEN] == '0';

        elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 14 && CRm == 0 && opc2 == 0 then /
            trap = CNTKCTL[][ELOPTEN] == '0' && CNTKCTL[][ELOVCTEN] == '0';

        elseif op == SystemAccessType\_RRT && opc1 == 1 && CRm == 14 then /
            trap = CNTKCTL[][ELOVCTEN] == '0';

    if trap then
        AArch64.AArch32SystemAccessTrap(EL1, instr);
```



```

// AArch64.CheckAArch32SystemAccessEL2Traps()
// =====
// Check for configurable traps to EL2 of an AArch32 System register access instruction.

AArch64.CheckAArch32SystemAccessEL2Traps(bits(32) instr)
    assert EL2Enabled() && PSTATE.EL IN {EL0, EL1, EL2};

    trap = FALSE;

    // Decode the AArch32 System register access instruction
    (op, cp_num, opcl, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

    if cp_num == 14 && PSTATE.EL IN {EL0, EL1} then
        if ((op == SystemAccessType\_RT && opcl == 0 && CRn == 1 && CRm == 0 && opc2 == 0) || // DBGDRAR
            (op == SystemAccessType\_RRT && opcl == 0 && CRm == 1) || // DBGDRAR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 2 && CRm == 0 && opc2 == 0) || // DBGDSAR
            (op == SystemAccessType\_RRT && opcl == 0 && CRm == 2)) then // DBGDSAR
            trap = MDCR_EL2.TDRA == '1' || MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1';

        elseif ((op == SystemAccessType\_RT && opcl == 0 && CRn == 1 && CRm == 0 && opc2 == 4) || // DBGDRAR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 1 && CRm == 1 && opc2 == 4) || // DBGDRAR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 1 && CRm == 3 && opc2 == 4) || // DBGDRAR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 1 && CRm == 4 && opc2 == 4)) then // DBGDRAR
            trap = MDCR_EL2.TDOSA == '1' || MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1';

        elseif opcl == 0 && (!Halted()) || !(op == SystemAccessType\_RT && CRn == 0 && CRm == 5 && opc2 == 5) then
            trap = MDCR_EL2.TDA == '1' || MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1';

        elseif opcl == 1 then
            trap = CPTR_EL2.TTA == '1';

        elseif op == SystemAccessType\_RT && opcl == 7 && CRn == 0 && CRm == 0 && opc2 == 0 then // JIDR
            trap = HCR_EL2.TID0 == '1';

    elseif cp_num == 14 && PSTATE.EL == EL2 then
        if opcl == 1 then
            trap = CPTR_EL2.TTA == '1';

    elseif cp_num == 15 && PSTATE.EL IN {EL0, EL1} then
        if ((op == SystemAccessType\_RT && opcl == 0 && CRn == 1 && CRm == 0 && opc2 == 0) || // SCTLR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 2 && CRm == 0 && opc2 == 0) || // TTBR0
            (op == SystemAccessType\_RRT && opcl == 0 && CRm == 2) || // TTBR0
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 2 && CRm == 0 && opc2 == 1) || // TTBR1
            (op == SystemAccessType\_RRT && opcl == 1 && CRm == 2) || // TTBR1
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 2 && CRm == 0 && opc2 == 2) || // TTBCR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 2 && CRm == 0 && opc2 == 3) || // TTBCR2
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 3 && CRm == 0 && opc2 == 0) || // DACR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 5 && CRm == 0 && opc2 == 0) || // DFSR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 5 && CRm == 0 && opc2 == 1) || // IFSR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 6 && CRm == 0 && opc2 == 0) || // DFAR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 6 && CRm == 0 && opc2 == 2) || // IFAR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 5 && CRm == 1 && opc2 == 0) || // ADFSR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 5 && CRm == 1 && opc2 == 1) || // AIFSR
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 10 && CRm == 2 && opc2 == 0) || // PRRR/M
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 10 && CRm == 2 && opc2 == 1) || // NMRR/M
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 10 && CRm == 3 && opc2 == 0) || // AMAIR0
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 10 && CRm == 3 && opc2 == 1) || // AMAIR1
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 13 && CRm == 0 && opc2 == 1)) then // CON
            trap = if write then HCR_EL2.TVM == '1' else HCR_EL2.TVMM == '1';
        elseif op == SystemAccessType\_RT && opcl == 0 && CRn == 8 then // TLBI
            trap = write && HCR_EL2.TTLB == '1';
        elseif ((op == SystemAccessType\_RT && opcl == 0 && CRn == 7 && CRm == 6 && opc2 == 2) || // DCI
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 7 && CRm == 10 && opc2 == 2) || // DCC
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 7 && CRm == 14 && opc2 == 2)) then // DCC
            trap = write && HCR_EL2.TSW == '1';

        elseif ((op == SystemAccessType\_RT && opcl == 0 && CRn == 7 && CRm == 6 && opc2 == 1) || // DCI
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 7 && CRm == 10 && opc2 == 1) || // DCC
            (op == SystemAccessType\_RT && opcl == 0 && CRn == 7 && CRm == 14 && opc2 == 1)) then // DCC
            trap = write && HCR_EL2.TPCP == '1';

```

```

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 1) || // ICI
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 0) || // ICI
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 1 && opc2 == 0) || // ICI
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 11 && opc2 == 1)) then // DCC
    trap = write && HCR_EL2.TPU == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 1) || // AC
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 3)) then // AC
    trap = HCR_EL2.TACR == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 2) || // TCM
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 3) || // TLB
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 6) || // REV
        (op == SystemAccessType\_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 7)) then // AID
    trap = HCR_EL2.TID1 == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 1) || // CTR
        (op == SystemAccessType\_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 0) || // CCS
        (op == SystemAccessType\_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 2) || // CCS
        (op == SystemAccessType\_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 1) || // CLI
        (op == SystemAccessType\_RT && opc1 == 2 && CRn == 0 && CRm == 0 && opc2 == 0)) then // CSS
    trap = HCR_EL2.TID2 == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 1) || // ID_
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 2 && opc2 <= 7) || // ID_
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm >= 3 && opc2 <= 1) || // Res
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 3 && opc2 == 2) || // Res
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 IN {4,5})) then //
    trap = HCR_EL2.TID3 == '1';

elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 2 then // CPA
    trap = CPTR_EL2.TCPAC == '1';

elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0 then // PMO
    trap = MDCR_EL2.TPMCR == '1' || MDCR_EL2.TPM == '1';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 14 && CRm >= 8) || // PMEVCNTR<n>
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 9 && CRm IN {12,13,14}) || // PM*
        (op == SystemAccessType\_RRT && opc1 == 0 && CRm == 9)) then // PMCCNTR (MR
    trap = MDCR_EL2.TPM == '1';

elseif op == SystemAccessType\_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then
    if !HaveVirtHostExt() || HCR_EL2.E2H == '0' then
        trap = CNTHCTL_EL2.EL1PCEN == '0';
    else
        trap = CNTHCTL_EL2.EL1PTEN == '0';
elseif op == SystemAccessType\_RRT && opc1 == 0 && CRm == 14 then
    trap = CNTHCTL_EL2.EL1PCTEN == '0';

elseif ((op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 0) || // S
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 2) || // N
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 12 && CRm == 0 && opc2 == 1) || // M
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 1) || // S
        (op == SystemAccessType\_RT && opc1 == 0 && CRn == 7 && CRm == 8 && opc2 >= 4)) then // A
    trap = IsSecureEL2Enabled() && PSTATE.EL == EL1 && IsSecure() && ELUsingAArch32(EL1);

if trap then
    AArch64.AArch32SystemAccessTrap(EL2, instr);

```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccessEL3Traps

```
// AArch64.CheckAArch32SystemAccessEL3Traps()
// =====
// Check for configurable traps to EL3 of an AArch32 System register access instruction.

AArch64.CheckAArch32SystemAccessEL3Traps(bits(32) instr)
    assert HaveEL(EL3) && PSTATE.EL != EL3;

    // Decode the AArch32 System register access instruction
    (op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

    trap = FALSE;

    if cp_num == 14 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 4 && !write) ||
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 4 && write) ||
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 4) ||
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 4 && opc2 == 4)) then
            trap = MDCR_EL3.TDOSA == '1';

        elsif opc1 == 0 && (!Halted() || !(op == SystemAccessType_RT && CRn == 0 && CRm == 5 && opc2 == 4)) then
            trap = MDCR_EL3.TDA == '1';

        elsif opc1 == 1 then
            trap = CPTR_EL3.TTA == '1';

    elsif cp_num == 15 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 0) || // SCR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 2) || // NSAC
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 12 && CRm == 0 && opc2 == 1) || // MVBA
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 1) || // SDCR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 8 && opc2 >= 4)) then // ATS1
            trap = PSTATE.EL == EL1 && IsSecure();

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 2) || // CPAC
            (op == SystemAccessType_RT && opc1 == 4 && CRn == 1 && CRm == 1 && opc2 == 2)) then // HCPTR
            trap = CPTR_EL3.TCPAC == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8) || // PMEV
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm IN {12,13,14}) || // PM*
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 9)) then // PMCC
            trap = MDCR_EL3.TPM == '1';

    if trap then
        AArch64.AArch32SystemAccessTrap(EL3, instr);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccessTraps

```
// AArch64.CheckAArch32SystemAccessTraps()
// =====
// Check for configurable disables or traps to a higher EL of an AArch32 System register access.

AArch64.CheckAArch32SystemAccessTraps(bits(32) instr)

    if PSTATE.EL == EL0 then
        AArch64.CheckAArch32SystemAccessEL1Traps(instr);
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1, EL2} && !IsInHost() then
        AArch64.CheckAArch32SystemAccessEL2Traps(instr);
        AArch64.CheckAArch32SystemAccessEL3Traps(instr);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained AArch32 CP15 traps in HSTR_EL2 and HCR_EL2.

boolean AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} then
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR_EL2<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR_EL2.TIDCP
        if (HCR_EL2.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
             (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// =====
// Check against CPACR[]

AArch64.CheckFPAdvSIMDEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        case CPACR[].FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap(); // Also check against CPTR_EL2 and CPTR_EL3
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()

    if EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.FPEN of
                when 'x0' disabled = !(PSTATE.EL == EL1 && HCR_EL2.TGE == '1');
                when '01' disabled = (PSTATE.EL == EL0 && HCR_EL2.TGE == '1');
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

    return;
```



## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

```
// AArch64.CheckForERetTrap()
// =====
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

// Non-secure EL1 execution of ERET, ERETAA, ERETAB when HCR_EL2.NV bit is set, is trapped to EL2
route_to_el2 = HaveNVExt() && EL2Enabled() && PSTATE.EL == EL1 && HCR_EL2.NV == '1';

if route_to_el2 then
    ExceptionRecord exception;
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = ExceptionSyndrome(Exception_ERetTrap);
    if !eret_with_pac then // ERET
        exception.syndrome<1> = '0';
        exception.syndrome<0> = '0'; // RES0
    else
        exception.syndrome<1> = '1';
        if pac_uses_key_a then // ERETAA
            exception.syndrome<0> = '0';
        else // ERETAB
            exception.syndrome<0> = '1';
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSMCUnDefOrTrap

```
// AArch64.CheckForSMCUnDefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUnDefOrTrap(bits(16) imm)
    route_to_el2 = EL2Enabled() && PSTATE.EL == EL1 && HCR_EL2.TSC == '1';
    if PSTATE.EL == EL0 then UNDEFINED;
    if !then UnallocatedEncoding();
    if !HaveEL(EL3) then
        if EL2Enabled() && PSTATE.EL == EL1 then
            if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UnallocatedEncoding();
        else
            UnallocatedEncoding() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UNDEFINED;
        else
            UNDEFINED;
    ();
    else
        route_to_el2 = EL2Enabled() && PSTATE.EL == EL1 && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWfxTrap

```
// AArch64.CheckForWfxTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch64.CheckForWfxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    case target_el of
        when EL1 trap = (if is_wfe then SCTLR{}.nTWE else SCTLR{}.nTWI) == '0';
        when EL2 trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3 trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
    if trap then
        AArch64.WfxTrap(target_el, is_wfe);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = EL2Enabled() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;

        exception = ExceptionSyndrome(Exception\_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =====
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception\_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.SystemRegisterTrap

```
// AArch64.SystemRegisterTrap()
// =====
// Trapped system register access other than due to CPTR_EL2 and CPACR_EL1

AArch64.SystemRegisterTrap(bits(2) target_el, bits(2) op0, bits(3) op2, bits(3) op1, bits(4) crn,
                           bits(5) rt, bits(4) crm, bit dir)
    assert UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
    exception.syndrome<21:20> = op0;
    exception.syndrome<19:17> = op2;
    exception.syndrome<16:14> = op1;
    exception.syndrome<13:10> = crn;
    exception.syndrome<9:5>   = rt;
    exception.syndrome<4:1>   = crm;
    exception.syndrome<0>     = dir;

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.UndefinedFault

```
// AArch64.UndefinedFault()
// =====

AArch64.UndefinedFault()

    route_to_el2 = EL2Enabled() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =====

AArch64.WFxTrap(bits(2) target_el, boolean is_wfe)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<0> = if is_wfe then '1' else '0';

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =====
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled\(\);
```

## Library pseudocode for aarch64/functions/aborts/AArch64.CreateFaultRecord

```
// AArch64.CreateFaultRecord()
// =====

FaultRecord AArch64.CreateFaultRecord(Fault type, bits(52) ipaddress, bit NS,
                                       integer level, AccType acctype, boolean write, bit extflag,
                                       bits(2) errortype, boolean secondstage, boolean s2fslwalk)

    FaultRecord fault;
    fault.type = type;
    fault.domain = bits(4) UNKNOWN;           // Not used from AArch64
    fault.debugmoe = bits(4) UNKNOWN;         // Not used from AArch64
    fault.errortype = errortype;
    fault.ipaddress.NS = NS;
    fault.ipaddress.address = ipaddress;
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fslwalk = s2fslwalk;

    return fault;
```

## Library pseudocode for aarch64/functions/aborts/AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// =====
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception Level using AArch64.

bits(25) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.type != Fault\_None;

    bits(25) iss = Zeros\(\);
    if HaveRASExt\(\) && IsExternalSyncAbort(fault) then iss<12:11> = fault.errortype; // SET
    if d_side then
        if IsSecondStage(fault) && !fault.s2fslwalk then iss<24:14> = LSInstructionSyndrome\(\);
        if HaveNV2Ext\(\) && fault.acctype == AccType\_NV2REGISTER then
            iss<13> = '1'; // Value of '1' indicates fault is generated by use of VNCR_EL2
        if fault.acctype IN {AccType\_DC, AccType\_DC\_UNPRIV, AccType\_IC, AccType\_AT} then
            iss<8> = '1'; iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fslwalk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.type, fault.level);

    return iss;
```

## Library pseudocode for aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// =====

// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    return passed;
```

## Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

## Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

## Library pseudocode for aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// =====

// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)

    acctype = AccType\_ATOMIC;
    iswrite = FALSE;
    aligned = (address == Align(address, size));
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.address, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

## Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// =====

bits(N) FPRSqrtStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1, sign1, value1) = FPUnpack(op1, FPCR);
    (type2, sign2, value2) = FPUnpack(op2, FPCR);
    (done, result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType Infinity);
        inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero);
        zero2 = (type2 == FPType Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```

## Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// =====

bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1,sign1,value1) = FPUnpack(op1, FPCR);
    (type2,sign2,value2) = FPUnpack(op2, FPCR);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType Infinity);
        inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero);
        zero2 = (type2 == FPType Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0');
        elseif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```

## Library pseudocode for aarch64/functions/memory/AArch64.CheckAlignment

```
// AArch64.CheckAlignment()
// =====

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype,
                                boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic = acctype IN { AccType ATOMIC, AccType ATOMICRW, }, };
    ordered = acctype IN { AccType ORDEREDATOMIC, AccType ORDEREDATOMICRW };
    ordered = acctype IN { AccType ORDERED, AccType ORDEREDRW, AccType LIMITEDORDERED, AccType ORDEREDA
    vector = acctype == AccType VEC;
    if SCTLR[][.A] == '1' then check = TRUE;
    elseif HaveUA16Ext() then
        check = (UInt(address<0+:4>) + alignment > 16) && ((ordered && (address<0+:4>) + alignment >= 16
    else check = atomic || ordered;

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

## Library pseudocode for aarch64/functions/memory/AArch64.MemSingle

```
// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if accdesc = HaveMTEExtCreateAccessDescriptor() then
        if(acctype);
        value = \_Mem[memaddrdesc, size, accdesc];
        return value;

// AArch64.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes. AccessIsTagChecked(ZeroExtend(address, 64),
    bits(4) ptag = TransformTag(ZeroExtend(address, 64));
    if !CheckTag(memaddrdesc, ptag, iswrite) then
        TagCheckFail(ZeroExtend(address, 64), iswrite);
    value = \_Mem[memaddrdesc, size, accdesc];
    return value;

// AArch64.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8) v
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if accdesc = HaveMTEExtCreateAccessDescriptor() then
        if AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = TransformTag(ZeroExtend(address, 64));
            if !CheckTag(memaddrdesc, ptag, iswrite) then
                TagCheckFail(ZeroExtend(address, 64), iswrite);
(acctype);
    \_Mem[memaddrdesc, size, accdesc] = value;
    return;
```



## Library pseudocode for aarch64/functions/memory/AddressWithAllocationTag

```
// AddressWithAllocationTag()
// =====
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as â€™0000â€™.

bits(64) AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag = allocation_tag - ('000':address<55>);
    result<59:56> = tag;
    return result;
```

## Library pseudocode for aarch64/functions/memory/AllocationTagFromAddress

```
// AllocationTagFromAddress()
// =====
// Generate a Tag from a 64-bit value containing a Logical Address Tag.
// If access to Allocation Tags is disabled, this function returns â€™0000â€™.

bits(4) AllocationTagFromAddress(bits(64) tagged_address)
    bits(4) logical_tag = tagged_address<59:56>;
    bits(4) tag = logical_tag + ('000':tagged_address<55>);
    return tag;
```

## Library pseudocode for aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// =====
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];
    if PSTATE.EL == ELO then
        stack_align_check = (SCTLR[][.SA0] != '0');
    else
        stack_align_check = (SCTLR[][.SA] != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```



```

// CheckTag()
// =====
// Performs a Tag Check operation for a memory access and returns
// whether the check passed
// Mem[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

boolean bits(size*8) CheckTag(Mem[bits(64) address, integer size, AccType acctype, memaddrdesc, b
    if memaddrdesc.memattrs.tagged then
        bits(64) paddress = acctype;
        assert size IN {1, 2, 4, 8, 16};
        bits(size*8) value;
        integer i;
        boolean iswrite = FALSE;

        aligned = ZeroExtendAArch64.CheckAlignment(memaddrdesc.paddress.address);
        return ptag == (address, size, acctype, iswrite);
        if size != 16 || !(acctype IN { , AccType_VECSTREAM}) then
            atomic = aligned;
        else
            // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
            // 64-bit aligned.
            atomic = address == Align(address, 8);

        if !atomic then
            assert size > 1;
            value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

            // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
            // access will generate an Alignment Fault, as to get this far means the first byte did
            // not, so we must be changing to a new translation page.
            if !aligned then
                c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
                assert c IN {Constraint_FAULT, Constraint_NONE};
                if c == Constraint_NONE then aligned = TRUE;

            for i = 1 to size-1
                value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
            elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
                value<63:0> = AArch64.MemSingle[address, 8, acctype, aligned];
                value<127:64> = AArch64.MemSingle[address+8, 8, acctype, aligned];
            else
                value = AArch64.MemSingle[address, size, acctype, aligned];

            if (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '1') || BigEndian() then
                value = BigEndianReverse(value);
            return value;

// Mem[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
    integer i;
    boolean iswrite = TRUE;

    if (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '1') || BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then

```

```

assert size > 1;
AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

// For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
// access will generate an Alignment Fault, as to get this far means the first byte did
// not, so we must be changing to a new translation page.
if !aligned then
    c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
    assert c IN {Constraint_FAULT, Constraint_NONE};
    if c == Constraint_NONE then aligned = TRUE;

for i = 1 to size-1
    AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
elseif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
    AArch64.MemSingle[address, 8, acctype, aligned] = value<63:0>;
    AArch64.MemSingle[address+8, 8, acctype, aligned] = value<127:64>;
else
    AArch64.MemSingleMemTagAccType_VEC[paddress];
else
    return TRUE;[address, size, acctype, aligned] = value;
return;

```

```

// If the implementation supports changing the block size without a break-before-make
// approach, then for implementations that have level 1 or 2 support, the nT bit in
// the block descriptor is valid.
boolean// AddPAC()
// =====
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) IsBlockDescriptorNTBitValid(); AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
bits(64) PAC;
bits(64) result;
bits(64) ext_ptr;
bits(64) extfield;
bit selbit;
boolean tbi = CalculateTBI(ptr, data);
integer top_bit = if tbi then 55 else 63;

// If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
// the pointer to select between upper and lower ranges, and preserve this.
// This handles the awkward case where there is apparently no correct choice between
// the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
// and 0xxxxxxx1 with TBI1=0 and TBI0=1:
if PtrHasUpperAndLowerAddRanges() then
    assert S1TranslationRegime() IN {EL1, EL2};
    if S1TranslationRegime() == EL1 then
        // EL1 translation regime registers
        if data then
            selbit = if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then ptr<55> else ptr<63>;
        else
            if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                selbit = ptr<55>;
            else
                selbit = ptr<63>;
        else
            // EL2 translation regime registers
            if data then
                selbit = if ((HaveEL(EL2) && TCR_EL2.TBI1 == '1') ||
                    (HaveEL(EL2) && TCR_EL2.TBI0 == '1')) then ptr<55> else ptr<63>;
            else
                selbit = if ((HaveEL(EL2) && TCR_EL2.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (HaveEL(EL2) && TCR_EL2.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then ptr<55>
            else selbit = if tbi then ptr<55> else ptr<63>;

integer bottom_PAC_bit = CalculateBottomPACBit(ptr, selbit);

// The pointer authentication code field takes all the available bits in between
extfield = Replicate(selbit, 64);

// Compute the pointer authentication code for a ptr with good extension bits
if tbi then
    ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
else
    ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

// Check if the ptr has good extension bits and corrupt the pointer authentication code if not,
if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
    PAC<top_bit-1> = NOT(PAC<top_bit-1>);

// Preserve the determination between upper and lower address at bit<55> and insert PAC
if tbi then
    result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
else
    result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
return result;

```



```

// Mem[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    boolean iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then
        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

            for i = 1 to size-1
                value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
            elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
                value<63:0> = AArch64.MemSingle[address, 8, acctype, aligned];
                value<127:64> = AArch64.MemSingle[address+8, 8, acctype, aligned];
            else
                value = AArch64.MemSingle[address, size, acctype, aligned];

        if (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTL_EL2.EE == '1') || BigEndian() then
            value = BigEndianReverse(value);
        return value;

// Mem[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
    boolean iswrite = TRUE;

    if (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTL_EL2.EE == '1') || BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};

```

```

        if c == Constraint\_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
        elsif size == 16 && acctype IN {AccType\_VEC, AccType\_VECSTREAM} then
            AArch64.MemSingle[address, 8, acctype, aligned] = value<63:0>;
            AArch64.MemSingle[address+8, 8, acctype, aligned] = value<127:64>;
        else
            AArch64.MemSingle[address, size, acctype, aligned] = value;
        return;

```

### Library pseudocode for aarch64/functions/memory/MemTag

```

// MemTag[] - non-assignment (read) form
// =====
// Load an Allocation Tag from memory.

bits(4) MemTag[bits(64) address]
    AddressDescriptor memaddrdesc;
    bits(4) value;
    iswrite = FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, AccType\_NORMAL, iswrite, TRUE, TAG_GRANULE);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Return the granule tag if tagging is enabled...
    if AllocationTagAccessIsEnabled() then
        return _MemTag[memaddrdesc];
    else
        // ...otherwise read tag as zero.
        return '0000';

// MemTag[] - assignment (write) form
// =====
// Store an Allocation Tag to memory.

MemTag[bits(64) address] = bits(4) value
    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // Stores of allocation tags must be aligned
    if address != Align(address, TAG_GRANULE) then
        boolean secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(AccType\_NORMAL, iswrite, secondstage));

    wasaligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(address, AccType\_NORMAL, iswrite, wasaligned, TAG_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    if AllocationTagAccessIsEnabled() then
        _MemTag[memaddrdesc] = value;

```

### Library pseudocode for aarch64/functions/memory/TransformTag

```

// TransformTag()
// =====
// Apply tag transformation rules.

bits(4) TransformTag(bits(64) vaddr)
    bits(4) vtag = vaddr<59:56>;
    bits(4) tagdelta = ZeroExtend(vaddr<55>);
    bits(4) ptag = vtag + tagdelta;
    return ptag;

```



## Library pseudocode for aarch64/functions/memory/boolean

```
// boolean AccessIsTagChecked()
// =====
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AccessIsTagChecked(bits(64) vaddr, AccType acctype)
    if PSTATE.M<4> == '1' then return FALSE;

    if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
        return FALSE;

    if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') t

        return FALSE;

    if !AllocationTagAccessIsEnabled() then
        return FALSE;

    if acctype IN {AccType IFETCH, AccType PTW} then
        return FALSE;

    if acctype == AccType_NV2REGISTER then
        return FALSE;

    if PSTATE.TCO=='1' then
        return FALSE;

    if IsNonTagCheckedInstruction() then
        return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/pac/addpac/AddPAC

```
// AddPAC()
// =====
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean tbi = CalculateTBI(ptr, data);
    integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                selbit = if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then ptr<55> else ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                // EL2 translation regime registers
                if data then
                    selbit = if ((HaveEL(EL2) && TCR_EL2.TBI1 == '1') ||
                        (HaveEL(EL2) && TCR_EL2.TBI0 == '1')) then ptr<55> else ptr<63>;
                else
                    selbit = if ((HaveEL(EL2) && TCR_EL2.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                        (HaveEL(EL2) && TCR_EL2.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then ptr<55>
                    else selbit = if tbi then ptr<55> else ptr<63>;

    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

    // The pointer authentication code field takes all the available bits in between
    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

    // Check if the ptr has good extension bits and corrupt the pointer authentication code if not;
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        PAC<top_bit-1> = NOT(PAC<top_bit-1>);

    // Preserve the determination between upper and lower address at bit<55> and insert PAC
    if tbi then
        result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    return result;
```

## Library pseudocode for aarch64/functions/pac/addpacda/AddPACDA

```
// AddPACDA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnDA else SCTLRL_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APDAKey_EL1, TRUE);
```

## Library pseudocode for aarch64/functions/pac/addpacdb/AddPACDB

```
// AddPACDB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnDB else SCTLRL_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APDBKey_EL1, TRUE);
```

## Library pseudocode for aarch64/functions/pac/addpacga/AddPACGA

```
// AddPACGA()
// =====
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of X, Y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;

    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return ComputePAC(X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0><63:32>:Zeros(32));
```

## Library pseudocode for aarch64/functions/pac/addpacia/AddPACIA

```
// AddPACIA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIA else SCTLRL_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APIAKey_EL1, FALSE);
```

## Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB

```
// AddPACIB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIB else SCTLRL_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APIBKey_EL1, FALSE);
```

## Library pseudocode for aarch64/functions/pac/auth/Auth

```
// Auth()
// =====
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit keynumber)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = CalculateTBI(ptr, data);
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    (ptr, ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    if tbi then
        original_ptr = ptr<63:56>:extfield<56-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<64-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
            result = original_ptr;
        else
            error_code = keynumber:NOT(keynumber);
            result = original_ptr<63:55>:error_code:original_ptr<52:0>;
    else
        if ((PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit>) &&
            (PAC<63:56> == ptr<63:56>)) then
            result = original_ptr;
        else
            error_code = keynumber:NOT(keynumber);
            result = original_ptr<63>:error_code:original_ptr<60:0>;
    return result;
```



## Library pseudocode for aarch64/functions/pac/authda/AuthDA

```
// AuthDA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnDA else SCTLRL_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return Auth(X, Y, APDAKey_EL1, TRUE, '0');
```

## Library pseudocode for aarch64/functions/pac/authdb/AuthDB

```
// AuthDB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnDB else SCTLRL_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return Auth(X, Y, APDBKey_EL1, TRUE, '1');
```

## Library pseudocode for aarch64/functions/pac/authia/AuthIA

```
// AuthIA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIA else SCTLRL_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return Auth(X, Y, APIAKey_EL1, FALSE, '0');
```

## Library pseudocode for aarch64/functions/pac/authib/AuthIB

```
// AuthIB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIB else SCTLRL_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return Auth(X, Y, APIBKey_EL1, FALSE, '1');
```

## Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =====

integer CalculateBottomPACBit(bit top_bit)
CalculateBottomPACBit(bits(64) ptr, bit top_bit)
    integer tsz_field;

    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
            using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
        else
            // EL2 translation regime registers
            assert HaveEL(EL2);
            tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
            using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
    else
        tsz_field = if PSTATE.EL == EL2 then UInt(TCR_EL2.T0SZ) else UInt(TCR_EL3.T0SZ);
        using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';

    max_limit_tsz_field = (if !HaveSmallPageTblExt() then 39 else if using64k then 47 else 48);
    if tsz_field > max_limit_tsz_field then
        // TCR_ELx.TySZ is out of range
        c = ConstrainUnpredictable(Unpredictable.RESTnSZ);
        assert c IN {Constraint.FORCE, Constraint.NONE};
        if c == Constraint.FORCE then tsz_field = max_limit_tsz_field;
    tszmin = if using64k && VAMax() == 52 then 12 else 16;
    if tsz_field < tszmin then
        c = ConstrainUnpredictable(Unpredictable.RESTnSZ);
        assert c IN {Constraint.FORCE, Constraint.NONE};
        if c == Constraint.FORCE then tsz_field = tszmin;
    return (64-tsz_field);
```

## Library pseudocode for aarch64/functions/pac/calculatetbi/CalculateTBI

```
// CalculateTBI()
// =====

boolean CalculateTBI(bits(64) ptr, boolean data)
    boolean tbi = FALSE;

    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                tbi = if ptr<55> == '1' then TCR_EL1.TBI1 == '1' else TCR_EL1.TBI0 == '1';
            else
                if ptr<55> == '1' then
                    tbi = TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0';
                else
                    tbi = TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0';
        else
            // EL2 translation regime registers
            if data then
                tbi = if ptr<55> == '1' then TCR_EL2.TBI1 == '1' else TCR_EL2.TBI0 == '1';
            else
                if ptr<55> == '1' then
                    tbi = TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0';
                else
                    tbi = TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0';
    elseif PSTATE.EL == EL2 then
        tbi = if data then TCR_EL2.TBI=='1' else TCR_EL2.TBI=='1' && TCR_EL2.TBID=='0';
    elseif PSTATE.EL == EL3 then
        tbi = if data then TCR_EL3.TBI=='1' else TCR_EL3.TBI=='1' && TCR_EL3.TBID=='0';

    return tbi;
```

## Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```
array bits(64) RC[0..4];

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
    bits(64)    workingval;
    bits(64)    runningmod;
    bits(64)    roundkey;
    bits(64)    modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;
    RC[3] = 0x082EFA98EC4E6C89<63:0>;
    RC[4] = 0x452821E638D01377<63:0>;

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;
    for i = 0 to 4
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
            workingval = PACSub(workingval);
            runningmod = TweakShuffle(runningmod<63:0>);
        roundkey = modk0 EOR runningmod;
        workingval = workingval EOR roundkey;
        workingval = PACCellShuffle(workingval);
        workingval = PACMult(workingval);
        workingval = PACSub(workingval);
        workingval = PACCellShuffle(workingval);
        workingval = PACMult(workingval);
        workingval = key1 EOR workingval;
        workingval = PACCellInvShuffle(workingval);
        workingval = PACInvSub(workingval);
        workingval = PACMult(workingval);
        workingval = PACCellInvShuffle(workingval);
        workingval = workingval EOR key0;
        workingval = workingval EOR runningmod;
        for i = 0 to 4
            workingval = PACInvSub(workingval);
            if i < 4 then
                workingval = PACMult(workingval);
                workingval = PACCellInvShuffle(workingval);
            runningmod = TweakInvShuffle(runningmod<63:0>);
            roundkey = key1 EOR runningmod;
            workingval = workingval EOR RC[4-i];
            workingval = workingval EOR roundkey;
            workingval = workingval EOR Alpha;
        workingval = workingval EOR modk0;

    return workingval;
```

## Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```
// PACCellInvShuffle()
// =====

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

## Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

```
// PACCellShuffle()
// =====

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```



## Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// =====

bits(64) PACInvSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher

bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '0101';
        when '0001' Toutput<4*i+3:4*i> = '1110';
        when '0010' Toutput<4*i+3:4*i> = '1101';
        when '0011' Toutput<4*i+3:4*i> = '1000';
        when '0100' Toutput<4*i+3:4*i> = '1010';
        when '0101' Toutput<4*i+3:4*i> = '1011';
        when '0110' Toutput<4*i+3:4*i> = '0001';
        when '0111' Toutput<4*i+3:4*i> = '1001';
        when '1000' Toutput<4*i+3:4*i> = '0010';
        when '1001' Toutput<4*i+3:4*i> = '0110';
        when '1010' Toutput<4*i+3:4*i> = '1111';
        when '1011' Toutput<4*i+3:4*i> = '0000';
        when '1100' Toutput<4*i+3:4*i> = '0100';
        when '1101' Toutput<4*i+3:4*i> = '1100';
        when '1110' Toutput<4*i+3:4*i> = '0111';
        when '1111' Toutput<4*i+3:4*i> = '0011';
return Toutput;
```

## Library pseudocode for aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =====

bits(64) PACMult(bits(64) Sinput)
    bits(4) t0;
    bits(4) t1;
    bits(4) t2;
    bits(4) t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;
```

## Library pseudocode for aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// =====

bits(64) PACSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '1011';
        when '0001' Toutput<4*i+3:4*i> = '0110';
        when '0010' Toutput<4*i+3:4*i> = '1000';
        when '0011' Toutput<4*i+3:4*i> = '1111';
        when '0100' Toutput<4*i+3:4*i> = '1100';
        when '0101' Toutput<4*i+3:4*i> = '0000';
        when '0110' Toutput<4*i+3:4*i> = '1001';
        when '0111' Toutput<4*i+3:4*i> = '1110';
        when '1000' Toutput<4*i+3:4*i> = '0011';
        when '1001' Toutput<4*i+3:4*i> = '0111';
        when '1010' Toutput<4*i+3:4*i> = '0100';
        when '1011' Toutput<4*i+3:4*i> = '0101';
        when '1100' Toutput<4*i+3:4*i> = '1101';
        when '1101' Toutput<4*i+3:4*i> = '0010';
        when '1110' Toutput<4*i+3:4*i> = '0001';
        when '1111' Toutput<4*i+3:4*i> = '1010';
return Toutput;
```

## Library pseudocode for aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =====

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =====

bits(4) TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// =====

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =====

bits(64) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// =====

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
```

## Library pseudocode for aarch64/functions/pac/pac/HavePACExt

```
// HavePACExt()
// =====

boolean HavePACExt()
    return HasArchVersion(ARMv8p3);
```

## Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// =====

// Returns TRUE if the pointer has upper and lower address ranges

boolean PtrHasUpperAndLowerAddRanges()
    return PSTATE.EL == EL1 || PSTATE.EL == EL0 || (PSTATE.EL == EL2 && HCR_EL2.E2H == '1');
```

## Library pseudocode for aarch64/functions/pac/strip/Strip

```
// Strip()
// =====
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = CalculateTBI(A, data);
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    (A, A<55>);
    extfield = Replicate(A<55>, 64);

    if tbi then
        original_ptr = A<63:56>:extfield< 56-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield< 64-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;

    case PSTATE.EL of
        when EL0
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return original_ptr;
```

## Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// =====
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

```
// AArch64.ESBOperation()
// =====
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()

    route_to_el3 = (HaveEL(EL3) && SCR_EL3.EA == '1');
    route_to_el2 = (EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    target = (if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1);

    if target == EL1 then
        mask_active = (PSTATE.EL IN {EL0, EL1});
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = (PSTATE.EL IN {EL0, EL2});
    else
        mask_active = (PSTATE.EL == target);

    mask_set = (PSTATE.A == '1' && (!mask_set = (PSTATE.A == '1')));
    intdis = (HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
              PSTATE.EL != EL3 || SCR_EL3.NMEA == '0');
    intdis = (Halted() || ExternalDebugInterruptsDisabled(target));
    masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending
    if IsPhysicalSErrorPending() && masked then
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            syndrome32 = AArch32.PhysicalSErrorSyndrome();
            DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        else
            implicit_esb = FALSE;
            syndrome64 = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = AArch64.ReportDeferredSError(syndrome64);
            ClearPendingPhysicalSError(); // Set ISR_EL1.A to 0
    (+)

return;
```

## Library pseudocode for aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

```
// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);
```

## Library pseudocode for aarch64/functions/ras/AArch64.ReportDeferredSError

```
// AArch64.ReportDeferredSError()
// =====
// Generate deferred SError syndrome

bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31> = '1'; // A
    target<24> = syndrome<24>; // IDS
    target<23:0> = syndrome<23:0>; // ISS
    return target;
```

## Library pseudocode for aarch64/functions/ras/AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =====
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
    assert EL2Enabled\(\) && PSTATE.EL IN {EL0,EL1};

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis      = Halted\(\) || ExternalDebugInterruptsDisabled\(EL1\);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if ELUsingAArch32\(EL1\) then
            VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        else
            VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>);
            HCR_EL2.VSE = '0'; // Clear pending virtual SError
            HCR_EL2.VSE = '0';

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```
// AArch64.MaybeZeroRegisterUppers()
// =====
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32\(\); // Always called from AArch32 state before entering AArch64 state

    if PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\) then
        first = 0; last = 14; include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0,EL1} && EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        first = 0; last = 30; include_R15 = FALSE;
    else
        first = 0; last = 30; include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[n]<63:32> = Zeros\(\);

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// =====

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i] = bits(64) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// =====

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// =====

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if HaveEL\(EL2\) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(32) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;
    if HaveEL\(EL3\) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(32) UNKNOWN;
        ELR_EL3 = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL\(EL1\) then
        SPSR_fiq = bits(32) UNKNOWN;
        SPSR_irq = bits(32) UNKNOWN;
        SPSR_abt = bits(32) UNKNOWN;
        SPSR_und = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetSystemRegisters

```
AArch64.ResetSystemRegisters(boolean cold_reset);
```

## Library pseudocode for aarch64/functions/registers/PC

```
// PC - non-assignment form
// =====
// Read program counter.

bits(64) PC[]
    return _PC;
```

## Library pseudocode for aarch64/functions/registers/SP

```
// SP[] - assignment form
// =====
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
    assert width IN {32,64};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value);
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = ZeroExtend(value);
            when EL1  SP_EL1 = ZeroExtend(value);
            when EL2  SP_EL2 = ZeroExtend(value);
            when EL3  SP_EL3 = ZeroExtend(value);
        return;

// SP[] - non-assignment form
// =====
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
    assert width IN {8,16,32,64};
    if PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0<width-1:0>;
            when EL1  return SP_EL1<width-1:0>;
            when EL2  return SP_EL2<width-1:0>;
            when EL3  return SP_EL3<width-1:0>;
```

## Library pseudocode for aarch64/functions/registers/V

```
// V[] - assignment form
// =====

V[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    integer vlen = if IsSVEEnabled(PSTATE.EL) then VL else 128;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value);
    else
        _Z[n]<vlen-1:0> = ZeroExtend(value);

// V[] - non-assignment form
// =====

bits(width) V[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _Z[n]<width-1:0>;
```



## Library pseudocode for aarch64/functions/registers/Vpart

```
// Vpart[] - non-assignment form
// =====

bits(width) Vpart(integer n, integer part]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        return V[n];
    else
        assert width == 64;
        return _V[n]<(width * 2)-1:width>;

// Vpart[] - assignment form
// =====

Vpart(integer n, integer part] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        V[n] = value;
    else
        assert width == 64;
        bits(64) vreg = V[n];
        V[n] = value<63:0> : vreg;
```

## Library pseudocode for aarch64/functions/registers/X

```
// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value.

X(integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value);
    return;

// X[] - non-assignment form
// =====
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X(integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
```

## Library pseudocode for aarch64/functions/sve/AArch32.IsFPEnabled

```
// AArch32.IsFPEnabled()
// =====

boolean AArch32.IsFPEnabled(bits(2) el)
    if el == EL0 && !ELUsingAArch32(EL1) then
        return AArch64.IsFPEnabled(el);

    if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
        // Check if access disabled in NSACR
        if NSACR.cp10 == '0' then return FALSE;

    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        case CPACR.cp10 of
            when 'x0' disabled = TRUE;
            when '01' disabled = (el == EL0);
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    if el IN {EL0, EL1, EL2} then
        if EL2Enabled() then
            if !ELUsingAArch32(EL2) then
                if CPTR_EL2.TFP == '1' then return FALSE;
            else
                if HCPTR.TCP10 == '1' then return FALSE;

    if HaveEL(EL3) && !ELUsingAArch32(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/AArch64.IsFPEnabled

```
// AArch64.IsFPEnabled()
// =====

boolean AArch64.IsFPEnabled(bits(2) el)
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} then
        // Check FP&SIMD at EL0/EL1
        case CPACR[].FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = (el == EL0);
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            if CPTR_EL2.FPEN == 'x0' then return FALSE;
        else
            if CPTR_EL2.TFP == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/CeilPow2

```
// CeilPow2()
// =====

// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 2;
    return FloorPow2(x - 1) * 2;
```

## Library pseudocode for aarch64/functions/sve/CheckSVEEnabled

```
// CheckSVEEnabled()
// =====

CheckSVEEnabled()
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} then
        // Check SVE at EL0/EL1
        case CPACR[][ZEN] of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL1);

    // Check FP&SIMD at EL0/EL1
    case CPACR[][FPEN] of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0;
        when '11' disabled = FALSE;
    if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            if CPTR_EL2.ZEN == 'x0' then SVEAccessTrap(EL2);
            if CPTR_EL2.FPEN == 'x0' then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then SVEAccessTrap(EL3);
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
```

## Library pseudocode for aarch64/functions/sve/DecodePredCount

```
// DecodePredCount()
// =====

integer DecodePredCount(bits(5) pattern, integer esize)
    integer elements = VL DIV esize;
    integer numElem;
    case pattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise    numElem = 0;
    return numElem;
```

## Library pseudocode for aarch64/functions/sve/ElemFFR

```
// ElemFFR[] - non-assignment form
// =====

bit ElemFFR[integer e, integer esize]
    return ElemP[_FFR, e, esize];

// ElemFFR[] - assignment form
// =====

ElemFFR[integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= PL;
    _FFR<n+psize-1:n> = ZeroExtend(value, psize);
    return;
```

## Library pseudocode for aarch64/functions/sve/ElemP

```
// ElemP[] - non-assignment form
// =====

bit ElemP[bits(N) pred, integer e, integer esize]
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;

// ElemP[] - assignment form
// =====

ElemP[bits(N) &pred, integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= N;
    pred<n+psize-1:n> = ZeroExtend(value, psize);
    return;
```

## Library pseudocode for aarch64/functions/sve/FFR

```
// FFR[] - non-assignment form
// =====

bits(width) FFR[]
    assert width == PL;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// =====

FFR[] = bits(width) value
    assert width == PL;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
        _FFR = ZeroExtend(value);
    else
        _FFR<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sve/FPCompareNE

```
// FPCompareNE()
// =====

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPTYPE\_SNaN || type1==FPTYPE\_QNaN || type2==FPTYPE\_SNaN || type2==FPTYPE\_QNaN then
        result = TRUE;
    if type1==FPTYPE\_SNaN || type2==FPTYPE\_SNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
    return result;
```

## Library pseudocode for aarch64/functions/sve/FPCompareUN

```
// FPCompareUN()
// =====

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPTYPE\_SNaN || type2==FPTYPE\_SNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);
    return (type1==FPTYPE\_SNaN || type1==FPTYPE\_QNaN || type2==FPTYPE\_SNaN || type2==FPTYPE\_QNaN);
```

## Library pseudocode for aarch64/functions/sve/FPConvertSVE

```
// FPConvertSVE()
// =====

bits(M) FPConvertSVE(bits(N) op, FPCRTType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding);

// FPConvertSVE()
// =====

bits(M) FPConvertSVE(bits(N) op, FPCRTType fpcr)
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

## Library pseudocode for aarch64/functions/sve/FPExpA

```
// FPExpA()
// =====

bits(N) FPExpA(bits(N) op, FPCRTType fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx];
    if N == 16 then
        result<15:0> = '0':op<9:5>:coeff<9:0>;
    elsif N == 32 then
        result<31:0> = '0':op<13:6>:coeff<22:0>;
    else // N == 64
        result<63:0> = '0':op<16:6>:coeff<51:0>;

    return result;
```



```

// FPExpCoefficient()
// =====

bits(N) FPExpCoefficient[integer index]
  assert N IN {16,32,64};
  integer result;

  if N == 16 then
    case index of
      when 0 result = 0x0000;
      when 1 result = 0x0016;
      when 2 result = 0x002d;
      when 3 result = 0x0045;
      when 4 result = 0x005d;
      when 5 result = 0x0075;
      when 6 result = 0x008e;
      when 7 result = 0x00a8;
      when 8 result = 0x00c2;
      when 9 result = 0x00dc;
      when 10 result = 0x00f8;
      when 11 result = 0x0114;
      when 12 result = 0x0130;
      when 13 result = 0x014d;
      when 14 result = 0x016b;
      when 15 result = 0x0189;
      when 16 result = 0x01a8;
      when 17 result = 0x01c8;
      when 18 result = 0x01e8;
      when 19 result = 0x0209;
      when 20 result = 0x022b;
      when 21 result = 0x024e;
      when 22 result = 0x0271;
      when 23 result = 0x0295;
      when 24 result = 0x02ba;
      when 25 result = 0x02e0;
      when 26 result = 0x0306;
      when 27 result = 0x032e;
      when 28 result = 0x0356;
      when 29 result = 0x037f;
      when 30 result = 0x03a9;
      when 31 result = 0x03d4;

    elsif N == 32 then
      case index of
        when 0 result = 0x000000;
        when 1 result = 0x0164d2;
        when 2 result = 0x02cd87;
        when 3 result = 0x043a29;
        when 4 result = 0x05aac3;
        when 5 result = 0x071f62;
        when 6 result = 0x08980f;
        when 7 result = 0x0a14d5;
        when 8 result = 0x0b95c2;
        when 9 result = 0x0dladf;
        when 10 result = 0x0ea43a;
        when 11 result = 0x1031dc;
        when 12 result = 0x11c3d3;
        when 13 result = 0x135a2b;
        when 14 result = 0x14f4f0;
        when 15 result = 0x16942d;
        when 16 result = 0x1837f0;
        when 17 result = 0x19e046;
        when 18 result = 0x1b8d3a;
        when 19 result = 0x1d3eda;
        when 20 result = 0x1ef532;
        when 21 result = 0x20b051;
        when 22 result = 0x227043;
        when 23 result = 0x243516;
        when 24 result = 0x25fed7;
        when 25 result = 0x27cd94;

```



```

when 26 result = 0x29a15b;
when 27 result = 0x2b7a3a;
when 28 result = 0x2d583f;
when 29 result = 0x2f3b79;
when 30 result = 0x3123f6;
when 31 result = 0x3311c4;
when 32 result = 0x3504f3;
when 33 result = 0x36fd92;
when 34 result = 0x38fbaf;
when 35 result = 0x3aff5b;
when 36 result = 0x3d08a4;
when 37 result = 0x3f179a;
when 38 result = 0x412c4d;
when 39 result = 0x4346cd;
when 40 result = 0x45672a;
when 41 result = 0x478d75;
when 42 result = 0x49b9be;
when 43 result = 0x4bec15;
when 44 result = 0x4e248c;
when 45 result = 0x506334;
when 46 result = 0x52a81e;
when 47 result = 0x54f35b;
when 48 result = 0x5744fd;
when 49 result = 0x599d16;
when 50 result = 0x5bfbb8;
when 51 result = 0x5e60f5;
when 52 result = 0x60ccdf;
when 53 result = 0x633f89;
when 54 result = 0x65b907;
when 55 result = 0x68396a;
when 56 result = 0x6ac0c7;
when 57 result = 0x6d4f30;
when 58 result = 0x6fe4ba;
when 59 result = 0x728177;
when 60 result = 0x75257d;
when 61 result = 0x77d0df;
when 62 result = 0x7a83b3;
when 63 result = 0x7d3e0c;

else // N == 64
  case index of
    when 0 result = 0x00000000000000;
    when 1 result = 0x02C9A3E778061;
    when 2 result = 0x059B0D3158574;
    when 3 result = 0x0874518759BC8;
    when 4 result = 0x0B5586CF9890F;
    when 5 result = 0x0E3EC32D3D1A2;
    when 6 result = 0x11301D0125B51;
    when 7 result = 0x1429AAEA92DE0;
    when 8 result = 0x172B83C7D517B;
    when 9 result = 0x1A35BEB6FCB75;
    when 10 result = 0x1D4873168B9AA;
    when 11 result = 0x2063B88628CD6;
    when 12 result = 0x2387A6E756238;
    when 13 result = 0x26B4565E27CDD;
    when 14 result = 0x29E9DF51FDEE1;
    when 15 result = 0x2D285A6E4030B;
    when 16 result = 0x306FE0A31B715;
    when 17 result = 0x33C08B26416FF;
    when 18 result = 0x371A7373AA9CB;
    when 19 result = 0x3A7DB34E59FF7;
    when 20 result = 0x3DEA64C123422;
    when 21 result = 0x4160A21F72E2A;
    when 22 result = 0x44E086061892D;
    when 23 result = 0x486A2B5C13CD0;
    when 24 result = 0x4BFDAD5362A27;
    when 25 result = 0x4F9B2769D2CA7;
    when 26 result = 0x5342B569D4F82;
    when 27 result = 0x56F4736B527DA;
    when 28 result = 0x5AB07DD485429;

```

```

when 29 result = 0x5E76F15AD2148;
when 30 result = 0x6247EB03A5585;
when 31 result = 0x6623882552225;
when 32 result = 0x6A09E667F3BCD;
when 33 result = 0x6DFB23C651A2F;
when 34 result = 0x71F75E8EC5F74;
when 35 result = 0x75FEB564267C9;
when 36 result = 0x7A11473EB0187;
when 37 result = 0x7E2F336CF4E62;
when 38 result = 0x82589994CCE13;
when 39 result = 0x868D99B4492ED;
when 40 result = 0x8ACE5422AA0DB;
when 41 result = 0x8F1AE99157736;
when 42 result = 0x93737B0CDC5E5;
when 43 result = 0x97D829FDE4E50;
when 44 result = 0x9C49182A3F090;
when 45 result = 0xA0C667B5DE565;
when 46 result = 0xA5503B23E255D;
when 47 result = 0xA9E6B5579FDBF;
when 48 result = 0xAE89F995AD3AD;
when 49 result = 0xB33A2B84F15FB;
when 50 result = 0xB7F76F2FB5E47;
when 51 result = 0xBCC1E904BC1D2;
when 52 result = 0xC199BDD85529C;
when 53 result = 0xC67F12E57D14B;
when 54 result = 0xCB720DCEF9069;
when 55 result = 0xD072D4A07897C;
when 56 result = 0xD5818DCFBA487;
when 57 result = 0xDA9E603DB3285;
when 58 result = 0xDFC97337B9B5F;
when 59 result = 0xE502EE78B3FF6;
when 60 result = 0xEA4AFA2A490DA;
when 61 result = 0xEFA1BEE615A27;
when 62 result = 0xF50765B6E4540;
when 63 result = 0xFA7C1819E90D8;

```

```
return result<N-1:0>;
```

### Library pseudocode for aarch64/functions/sve/FPMinNormal

```

// FPMinNormal()
// =====

bits(N) FPMinNormal(bit sign)
  assert N IN {16,32,64};
  constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
  constant integer F = N - (E + 1);
  exp = Zeros(E-1):'1';
  frac = Zeros(F);
  return sign : exp : frac;

```

### Library pseudocode for aarch64/functions/sve/FPOne

```

// FPOne()
// =====

bits(N) FPOne(bit sign)
  assert N IN {16,32,64};
  constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
  constant integer F = N - (E + 1);
  exp = '0':Ones(E-1);
  frac = Zeros(F);
  return sign : exp : frac;

```

## Library pseudocode for aarch64/functions/sve/FPPointFive

```
// FPPointFive()
// =====

bits(N) FPPointFive(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-2):'0';
    frac = Zeros(F);
    return sign : exp : frac;
```

## Library pseudocode for aarch64/functions/sve/FPProcess

```
// FPProcess()
// =====

bits(N) FPProcess(bits(N) input)
    bits(N) result;
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(input, FPCR);
    if type == FPType\_SNaN || type == FPType\_QNaN then
        result = FPProcessNaN(type, input, FPCR);
    elsif type == FPType\_Infinity then
        result = FPInfinity(sign);
    elsif type == FPType\_Zero then
        result = FPZero(sign);
    else
        result = FPRound(value, FPCR);
    return result;
```

## Library pseudocode for aarch64/functions/sve/FPScale

```
// FPScale()
// =====

bits(N) FPScale(bits(N) op, integer scale, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPType\_SNaN || type == FPType\_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPType\_Zero then
        result = FPZero(sign);
    elsif type == FPType\_Infinity then
        result = FPInfinity(sign);
    else
        result = FPRound(value * (2.0scale), fpcr);
    return result;
```

## Library pseudocode for aarch64/functions/sve/FPTrigMAdd

```
// FPTrigMAdd()
// =====

bits(N) FPTrigMAdd(integer x, bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    assert x >= 0;
    assert x < 8;
    bits(N) coeff;

    if op2<N-1> == '1' then
        x = x + 8;
    op2<N-1> = '0';

    coeff = FPTrigMAddCoefficient[x];
    result = FPMulAdd(coeff, op1, op2, fpcr);

    return result;
```

## Library pseudocode for aarch64/functions/sve/FPTrigMAddCoefficient

```
// FPTrigMAddCoefficient()
// =====

bits(N) FPTrigMAddCoefficient[integer index]
  assert N IN {16,32,64};
  integer result;

  if N == 16 then
    case index of
      when 0 result = 0x3c00;
      when 1 result = 0xb155;
      when 2 result = 0x2030;
      when 3 result = 0x0000;
      when 4 result = 0x0000;
      when 5 result = 0x0000;
      when 6 result = 0x0000;
      when 7 result = 0x0000;
      when 8 result = 0x3c00;
      when 9 result = 0xb800;
      when 10 result = 0x293a;
      when 11 result = 0x0000;
      when 12 result = 0x0000;
      when 13 result = 0x0000;
      when 14 result = 0x0000;
      when 15 result = 0x0000;
    elseif N == 32 then
      case index of
        when 0 result = 0x3f800000;
        when 1 result = 0xbe2aaaab;
        when 2 result = 0x3c088886;
        when 3 result = 0xb95008b9;
        when 4 result = 0x36369d6d;
        when 5 result = 0x00000000;
        when 6 result = 0x00000000;
        when 7 result = 0x00000000;
        when 8 result = 0x3f800000;
        when 9 result = 0xbf000000;
        when 10 result = 0x3d2aaaa6;
        when 11 result = 0xbab60705;
        when 12 result = 0x37cd37cc;
        when 13 result = 0x00000000;
        when 14 result = 0x00000000;
        when 15 result = 0x00000000;
      else // N == 64
        case index of
          when 0 result = 0x3ff0000000000000;
          when 1 result = 0xbfc5555555555543;
          when 2 result = 0x3f8111111110f30c;
          when 3 result = 0xbf2a01a019b92fc6;
          when 4 result = 0x3ec71de351f3d22b;
          when 5 result = 0xbe5ae5e2b60f7b91;
          when 6 result = 0x3de5d8408868552f;
          when 7 result = 0x0000000000000000;
          when 8 result = 0x3ff0000000000000;
          when 9 result = 0xbfe0000000000000;
          when 10 result = 0x3fa5555555555536;
          when 11 result = 0xbf56c16c16c13a0b;
          when 12 result = 0x3efa01a019b1e8d8;
          when 13 result = 0xbe927e4f7282f468;
          when 14 result = 0x3e21ee96d2641b13;
          when 15 result = 0xbda8f76380fbb401;

  return result<N-1:0>;
```

## Library pseudocode for aarch64/functions/sve/FPTrigSMul

```
// FPTrigSMul()
// =====

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    result = FPMul(op1, op1, fpcr);
    (type, sign, value) = FPUnpack(result, fpcr);
    if (type != FPType\_QNaN) && (type != FPType\_SNaN) then
        result<N-1> = op2<0>;

    return result;
```

## Library pseudocode for aarch64/functions/sve/FPTrigSSel

```
// FPTrigSSel()
// =====

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};
    bits(N) result;

    if op2<0> == '1' then
        result = FPOne(op2<1>);
    else
        result = op1;
        result<N-1> = result<N-1> EOR op2<1>;

    return result;
```

## Library pseudocode for aarch64/functions/sve/FirstActive

```
// FirstActive()
// =====

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';
```

## Library pseudocode for aarch64/functions/sve/FloorPow2

```
// FloorPow2()
// =====
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
    assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);
```

## Library pseudocode for aarch64/functions/sve/HaveSVE

```
// HaveSVE()
// =====

boolean HaveSVE()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Have SVE ISA";
```

## Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

```
// ImplementedSVEVectorLength()
// =====
// Reduce SVE vector length to a supported value (e.g. power of two)

integer ImplementedSVEVectorLength(integer nbits)
    return integer IMPLEMENTATION_DEFINED;
```

## Library pseudocode for aarch64/functions/sve/IsEven

```
// IsEven()
// =====

boolean IsEven(integer val)
    return val MOD 2 == 0;
```

## Library pseudocode for aarch64/functions/sve/IsFPEnabled

```
// IsFPEnabled()
// =====

boolean IsFPEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return AArch32.IsFPEnabled(el);
    else
        return AArch64.IsFPEnabled(el);
```

## Library pseudocode for aarch64/functions/sve/IsSVEEnabled

```
// IsSVEEnabled()
// =====

boolean IsSVEEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} then
        // Check SVE at EL0/EL1
        case CPACR[].ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = (el == EL0);
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            if CPTR_EL2.ZEN == 'x0' then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/LastActive

```
// LastActive()
// =====

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';
```

## Library pseudocode for aarch64/functions/sve/LastActiveElement

```
// LastActiveElement()
// =====

integer LastActiveElement(bits(N) mask, integer esize)
    assert esize IN {8, 16, 32, 64};
    integer elements = VL DIV esize;
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return e;
    return -1;
```

## Library pseudocode for aarch64/functions/sve/MAX\_PL

```
constant integer MAX_PL = 256;
```

## Library pseudocode for aarch64/functions/sve/MAX\_VL

```
constant integer MAX_VL = 2048;
```

## Library pseudocode for aarch64/functions/sve/MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// =====

MaybeZeroSVEUppers(bits(2) target_el)
    boolean lower_enabled;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
        return;

    if target_el == EL3 then
        if EL2Enabled() then
            lower_enabled = IsFPEEnabled(EL2);
        else
            lower_enabled = IsFPEEnabled(EL1);
    else
        lower_enabled = IsFPEEnabled(target_el - 1);

    if lower_enabled then
        integer vl = if IsSVEEnabled(PSTATE.EL) then VL else 128;
        integer pl = vl DIV 8;
        for n = 0 to 31
            if ConstrainUnpredictableBool(Unpredictable_SVEZERoupper) then
                _Z[n] = ZeroExtend(_Z[n]<vl-1:0>);
        for n = 0 to 15
            if ConstrainUnpredictableBool(Unpredictable_SVEZERoupper) then
                _P[n] = ZeroExtend(_P[n]<pl-1:0>);
        if ConstrainUnpredictableBool(Unpredictable_SVEZERoupper) then
            _FFR = ZeroExtend(_FFR<pl-1:0>);
```



## Library pseudocode for aarch64/functions/sve/MemNF

```
// MemNF[] - non-assignment form
// =====

(bits(8*size), boolean) MemNF(bits(64) address, integer size, AccType acctype)
    assert size IN {1, 2, 4, 8, 16};
    bits(8*size) value;

    aligned = (address == Align(address, size));
    A = SCTLR[][.A];

    if !aligned && (A == '1') then
        return (bits(8*size) UNKNOWN, TRUE);

    atomic = aligned || size == 1;

    if !atomic then
        (value<7:0>, bad) = MemSingleNF[address, 1, acctype, aligned];

        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
            assert c IN {Constraint\_FAULT, Constraint\_NONE};
            if c == Constraint\_NONE then aligned = TRUE;

        for i = 1 to size-1
            (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, acctype, aligned];

            if bad then
                return (bits(8*size) UNKNOWN, TRUE);
    else
        (value, bad) = MemSingleNF[address, size, acctype, aligned];
        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

    if BigEndian() then
        value = BigEndianReverse(value);

    return (value, FALSE);
```

## Library pseudocode for aarch64/functions/sve/MemSingleNF

```
// MemSingleNF[] - non-assignment form
// =====

(bits(8*size), boolean) MemSingleNF(bits(64) address, integer size, AccType acctype, boolean wasaligned,
bits(8*size) value;
boolean iswrite = FALSE;
AddressDescriptor memaddrdesc;

// Implementation may suppress NF load for any reason
if ConstrainUnpredictableBool(Unpredictable\_NONFAULT) then
    return (bits(8*size) UNKNOWN, TRUE);

// MMU or MPU
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

// Non-fault load from Device memory must not be performed externally
if memaddrdesc.memattrs.type == MemType\_Device then
    return (bits(8*size) UNKNOWN, TRUE);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return (bits(8*size) UNKNOWN, TRUE);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if accdesc = HaveMTEExtCreateAccessDescriptor() then
    if AccessIsTagChecked(address, acctype) then
        bits(4) ptag = TransformTag(address);
        if !CheckTag(memaddrdesc, ptag, iswrite) then
            return (bits(8*size) UNKNOWN, TRUE);
(acctype);
value = _Mem[memaddrdesc, size, accdesc];

return (value, FALSE);
```

## Library pseudocode for aarch64/functions/sve/NoneActive

```
// NoneActive()
// =====

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
integer elements = N DIV (esize DIV 8);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' && ElemP[x, e, esize] == '1' then return '0';
return '1';
```

## Library pseudocode for aarch64/functions/sve/P

```
// P[] - non-assignment form
// =====

bits(width) P[integer n]
    assert n >= 0 && n <= 31;
    assert width == PL;
    return _P[n]<width-1:0>;

// P[] - assignment form
// =====

P[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == PL;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
        _P[n] = ZeroExtend(value);
    else
        _P[n]<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sve/PL

```
// PL - non-assignment form
// =====

integer PL
    return VL DIV 8;
```

## Library pseudocode for aarch64/functions/sve/PredTest

```
// PredTest()
// =====

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;
```

## Library pseudocode for aarch64/functions/sve/ReducePredicated

```
// ReducePredicated()
// =====

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    integer p2bits = CeilPow2(N);
    bits(p2bits) operand;
    integer elements = p2bits DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ElemP[mask, e, esize] == '1' then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return Reduce(op, operand, esize);
```

## Library pseudocode for aarch64/functions/sve/Reverse

```
// Reverse()
// =====
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    bits(N) result;
    integer sw = N DIV M;
    assert N == sw * M;
    for s = 0 to sw-1
        Elem[result, sw - 1 - s, M] = Elem[word, s, M];
    return result;
```

## Library pseudocode for aarch64/functions/sve/SVEAccessTrap

```
// SVEAccessTrap()
// =====
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    exception = ExceptionSyndrome(Exception\_SVEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/sve/SVECmp

```
enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```

## Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

```
// SVEMoveMaskPreferred()
// =====
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'ffffffffffffxy' or '0000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then
            return FALSE;

        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'ffxyffxyffxyffxy' or '00xy00xy00xy00xy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (IsZero(imm<15:7>) || IsOnes(imm<15:7>)) then
            return FALSE;

        // Check for 'xyxyxyxyxyxyxyxy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (imm<15:8> == imm<7:0>) then
            return FALSE;

    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffxy00' or '000000000000xy00'
        if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then
            return FALSE;

        // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'xy00xy00xy00xy00'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
            return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/System

```
array bits(MAX\_VL) _Z[0..31];
array bits(MAX\_PL) _P[0..15];
bits(MAX\_PL) _FFR;
```

## Library pseudocode for aarch64/functions/sve/VL

```
// VL - non-assignment form
// =====

integer VL
integer vl;

if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
    vl = UInt(ZCR_EL1.LEN);

if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
    vl = UInt(ZCR_EL2.LEN);
elseif EL2Enabled() && PSTATE.EL IN {EL0,EL1} then
    vl = Min(vl, UInt(ZCR_EL2.LEN));

if PSTATE.EL == EL3 then
    vl = UInt(ZCR_EL3.LEN);
elseif HaveEL(EL3) then
    vl = Min(vl, UInt(ZCR_EL3.LEN));

vl = (vl + 1) * 128;
vl = ImplementedSVEVectorLength(vl);

return vl;
```

## Library pseudocode for aarch64/functions/sve/Z

```
// Z[] - non-assignment form
// =====

bits(width) Z[integer n]
assert n >= 0 && n <= 31;
assert width == VL;
return _Z[n]<width-1:0>;

// Z[] - assignment form
// =====

Z[integer n] = bits(width) value
assert n >= 0 && n <= 31;
assert width == VL;
if ConstrainUnpredictableBool(Unpredictable\_SVEZERoupper) then
    _Z[n] = ZeroExtend(value);
else
    _Z[n]<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sysregisters/CNTKCTL

```
// CNTKCTL[] - non-assignment form
// =====

CNTKCTLType CNTKCTL[]
bits(32) r;
if IsInHost() then
    r = CNTHCTL_EL2;
    return r;
r = CNTKCTL_EL1;
return r;
```

## Library pseudocode for aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

## Library pseudocode for aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// =====

CPACRType CPACR[]
  bits(32) r;
  if IsInHost() then
    r = CPTR_EL2;
    return r;
  r = CPACR_EL1;
  return r;
```

## Library pseudocode for aarch64/functions/sysregisters/CPACRType

```
type CPACRType;
```

## Library pseudocode for aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// =====

bits(64) ELR[bits(2) el]
  bits(64) r;
  case el of
    when EL1  r = ELR_EL1;
    when EL2  r = ELR_EL2;
    when EL3  r = ELR_EL3;
    otherwise Unreachable();
  return r;

// ELR[] - non-assignment form
// =====

bits(64) ELR[]
  assert PSTATE.EL != EL0;
  return ELR[PSTATE.EL];

// ELR[] - assignment form
// =====

ELR[bits(2) el] = bits(64) value
  bits(64) r = value;
  case el of
    when EL1  ELR_EL1 = r;
    when EL2  ELR_EL2 = r;
    when EL3  ELR_EL3 = r;
    otherwise Unreachable();
  return;

// ELR[] - assignment form
// =====

ELR[] = bits(64) value
  assert PSTATE.EL != EL0;
  ELR[PSTATE.EL] = value;
  return;
```

## Library pseudocode for aarch64/functions/sysregisters/ESR

```
// ESR[] - non-assignment form
// =====

ESRType ESR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1    r = ESR_EL1;
        when EL2    r = ESR_EL2;
        when EL3    r = ESR_EL3;
        otherwise Unreachable();
    return r;

// ESR[] - non-assignment form
// =====

ESRType ESR[]
    return ESR\[S1TranslationRegime\]\(\);

// ESR[] - assignment form
// =====

ESR[bits(2) regime] = ESRType value
    bits(32) r = value;
    case regime of
        when EL1    ESR_EL1 = r;
        when EL2    ESR_EL2 = r;
        when EL3    ESR_EL3 = r;
        otherwise Unreachable();
    return;

// ESR[] - assignment form
// =====

ESR[] = ESRType value
    ESR\[S1TranslationRegime\]\(\) = value;
```

## Library pseudocode for aarch64/functions/sysregisters/ESRType

```
type ESRType;
```



## Library pseudocode for aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// =====

bits(64) FAR[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1   r = FAR_EL1;
    when EL2   r = FAR_EL2;
    when EL3   r = FAR_EL3;
    otherwise Unreachable();
  return r;

// FAR[] - non-assignment form
// =====

bits(64) FAR[]
  return FAR\[S1TranslationRegime\(\)\];

// FAR[] - assignment form
// =====

FAR[bits(2) regime] = bits(64) value
  bits(64) r = value;
  case regime of
    when EL1   FAR_EL1 = r;
    when EL2   FAR_EL2 = r;
    when EL3   FAR_EL3 = r;
    otherwise Unreachable();
  return;

// FAR[] - assignment form
// =====

FAR[] = bits(64) value
  FAR\[S1TranslationRegime\(\)\] = value;
  return;
```

## Library pseudocode for aarch64/functions/sysregisters/MAIR

```
// MAIR[] - non-assignment form
// =====

MAIRType MAIR[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1   r = MAIR_EL1;
    when EL2   r = MAIR_EL2;
    when EL3   r = MAIR_EL3;
    otherwise Unreachable();
  return r;

// MAIR[] - non-assignment form
// =====

MAIRType MAIR[]
  return MAIR\[S1TranslationRegime\(\)\];
```

## Library pseudocode for aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

## Library pseudocode for aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLR[bits(2) regime]
    bits(64) r;
    bits(32) r;
    case regime of
        when EL1 r = SCTLR_EL1;
        when EL2 r = SCTLR_EL2;
        when EL3 r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];
```

## Library pseudocode for aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

## Library pseudocode for aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// =====

bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = VBAR_EL1;
        when EL2 r = VBAR_EL2;
        when EL3 r = VBAR_EL3;
        otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// =====

bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
```

## Library pseudocode for aarch64/functions/system/AArch64.CheckAdvSIMDFPSystemRegisterTraps

```
// Checks if an AArch64 MSR, MRS or SYS instruction on a SIMD or floating-point
// register is trapped under the current configuration. Returns a boolean which
// is TRUE if trapping occurs, plus a binary value that specifies the Exception
// level trapped to.
(boolean, bits(2)) AArch64.CheckAdvSIMDFPSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bit
```

## Library pseudocode for aarch64/functions/system/AArch64.CheckSVESystemRegisterTraps

```
// Checks if an AArch64 MSR/MRS/SYS instruction on a Scalable Vector
// register is trapped under the current configuration
(boolean, bits(2)) AArch64.CheckSVESystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) c
```

## Library pseudocode for aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// =====

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bits(5) rt,
    // Checks if an AArch64 MSR, MRS or SYS instruction is UNALLOCATED or trapped at the current
    // exception level, security state and configuration, based on the opcode's encoding.
    boolean unallocated = FALSE;
    boolean need_secure = FALSE;
    bits(2) min_EL;

    // Check for traps by HCR_EL2.TIDCP
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && HCR_EL2.TIDCP == 1 && op0 == 'x1' && crn == '1x11' the
        // At EL0, it is IMPLEMENTATION_DEFINED whether attempts to execute system
        // register access instructions with reserved encodings are trapped to EL2 or UNDEFINED
        rcs_el0_trap = boolean IMPLEMENTATION_DEFINED "Reserved Control Space EL0 Trapped";
        if PSTATE.EL == EL1 || rcs_el0_trap then
            AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);

    // Check for unallocated encodings
    case op1 of
        when '00x', '010'
            min_EL = EL1;
        when '011'
            min_EL = EL0;
        when '100'
            min_EL = EL2;
        when '101'
            if !HaveVirtHostExt() then UNDEFINED;
            min_EL = () then UnallocatedEncoding();
            min_EL = EL2;
        when '110'
            min_EL = EL3;
        when '111'
            min_EL = EL1;
            need_secure = TRUE;

    if UInt(PSTATE.EL) < UInt(min_EL) then
        // Check for traps on read/write access to registers named _EL2, _EL02, _EL12 from non-secure E
        nv_access = HaveNVExt() && min_EL == EL2 && PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.NV == '
        if !nv_access then
            UNDEFINED;
        elsif need_secure && ! if !nv_access then UnallocatedEncoding();
        elsif need_secure && !IsSecure() then
            UnallocatedEncoding();
        elsif AArch64.CheckUnallocatedSystemAccess(PSTATE.EL, op0, op1, crn, crm, op2, read) then
            UnallocatedEncoding() then
            UNDEFINED;
        elsif AArch64.CheckUnallocatedSystemAccess(PSTATE.EL, op0, op1, crn, crm, op2, read) then
            UNDEFINED;
        ();

    // Check for traps on accesses to SIMD or floating-point registers
    (take_trap, target_el) = AArch64.CheckAdvSIMDFPSystemRegisterTraps(op0, op1, crn, crm, op2, read);
    if take_trap then
        AArch64.AdvSIMDFPAccessTrap(target_el);

    // Check for traps on accesses to Scalable Vector registers
    (take_trap, target_el) = AArch64.CheckSVESystemRegisterTraps(op0, op1, crn, crm, op2);
    if take_trap then
        SVEAccessTrap(target_el);

    // Check for traps on access to all other system registers
    (take_trap, target_el) = AArch64.CheckSystemRegisterTraps(op0, op1, crn, crm, op2, read);

    if take_trap then
        AArch64.SystemRegisterTrap(target_el, op0, op2, op1, crn, rt, crm, read);
```

### Library pseudocode for aarch64/functions/system/AArch64.CheckSystemRegisterTraps

```
// Checks if an AArch64 MSR, MRS or SYS instruction on a system register is trapped
// under the current configuration. Returns a boolean which is TRUE if trapping
// occurs, plus a binary value that specifies the Exception level trapped to.
(boolean, bits(2)) AArch64.CheckSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
```

### Library pseudocode for aarch64/functions/system/AArch64.CheckUnallocatedSystemAccess

```
// Checks if an AArch64 MSR, MRS or SYS instruction is unallocated under the current
// configuration.
boolean AArch64.CheckUnallocatedSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3)
```

### Library pseudocode for aarch64/functions/system/AArch64.ExecutingATS1xPInstr

```
// AArch64.ExecutingATS1xPInstr()
// =====
// Return TRUE if current instruction is AT S1E1R/WP

boolean AArch64.ExecutingATS1xPInstr()
    if !HavePrivATExt() then return FALSE;

    instr = ThisInstr();
    if instr<22+:10> == '1101010100' then
        op1 = instr<16+:3>;
        CRn = instr<12+:4>;
        CRm = instr<8+:4>;
        op2 = instr<5+:3>;
        return op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000', '001'};
    else
        return FALSE;
```

### Library pseudocode for aarch64/functions/system/AArch64.ExecutingBROrBLROrRetInstr

```
// AArch64.ExecutingBROrBLROrRetInstr()
// =====
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBROrBLROrRetInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;
```

### Library pseudocode for aarch64/functions/system/AArch64.ExecutingBTIInstr

```
// AArch64.ExecutingBTIInstr()
// =====
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
        CRm = instr<11:8>;
        op2 = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    else
        return FALSE;
```

## Library pseudocode for aarch64/functions/system/AArch64.SysInstr

```
// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

## Library pseudocode for aarch64/functions/system/AArch64.SysInstrWithResult

```
// Execute a system instruction with read (result operand).
// Returns the result of the instruction.
bits(64) AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2);
```

## Library pseudocode for aarch64/functions/system/AArch64.SysRegRead

```
// Read from a system register and return the contents of the register.
bits(64) AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2); System_Get
```

## Library pseudocode for aarch64/functions/system/AArch64.SysRegWrite

```
// Write to a system register.
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

## Library pseudocode for aarch64/functions/instrs/system/branch/BTypeCompatibleeret/AArch64.ExceptionReturn

```
boolean BTypeCompatible; // AArch64.ExceptionReturn()
// =====AArch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)

SynchronizeContext();

if HaveRASExt() && SCTLR[].IESB == '1' then
    SynchronizeErrors();
    iesb_req = TRUE;
    TakeUnmaskedPhysicalErrorInterrupts(iesb_req);
// Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
SetPSTATEFromPSR(spsr);
ClearExclusiveLocal(ProcessorID());
SendEventLocal();

if PSTATE.IL == '1' && spsr<4> == '1' && spsr<20> == '0' then
    // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN;
    new_pc<63:32> = bits(32) UNKNOWN;
    new_pc<1:0> = bits(2) UNKNOWN;
elseif UsingAArch32() then // Return to AArch32
    // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set s
    if PSTATE.T == '1' then
        new_pc<0> = '0'; // T32
    else
        new_pc<1:0> = '00'; // A32
    else // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc);

if UsingAArch32() then
    // 32 most significant bits are ignored.
    BranchTo(new_pc<31:0>, BranchType_UNKNOWN);
else
    BranchToAddr(new_pc, BranchType_ERET);
```

### Library pseudocode for aarch64/functions/system/BTypeCompatible\_BTI

```
// BTypeCompatible_BTI
// =====
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_BTI(bits(2) hintcode)
    case hintcode of
        when '00'
            return FALSE;
        when '01'
            return PSTATE.BTYPE != '11';
        when '10'
            return PSTATE.BTYPE != '10';
        when '11'
            return TRUE;
```

### Library pseudocode for aarch64/functions/system/BTypeCompatible\_PACIXSP

```
// BTypeCompatible_PACIXSP()
// =====
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
    if PSTATE.BTYPE IN {'01', '10'} then
        return TRUE;
    elsif PSTATE.BTYPE == '11' then
        index = if PSTATE.EL == ELO then 35 else 36;
        return SCTLR[]<index> == '0';
    else
        return FALSE;
```

### Library pseudocode for aarch64/functions/system/BTypeNext

```
bits(2) BTypeNext;
```

### Library pseudocode for aarch64/functions/system/InGuardedPage

```
boolean InGuardedPage;
```

## Library pseudocode for aarch64/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =====

AArch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalErrorInterrupts(iesb_req);
        // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
        SetPSTATEFromPSR(spsr);
        ClearExclusiveLocal(ProcessorID());
        SendEventLocal();

    if PSTATE.IL == '1' && spsr<4> == '1' && spsr<20> == '0' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set
        if PSTATE.T == '1' then
            new_pc<0> = '0'; // T32
        else
            new_pc<1:0> = '00'; // A32
    else // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc);

    if UsingAArch32() then
        // 32 most significant bits are ignored.
        BranchTo(new_pc<31:0>, BranchType ERET);
    else
        BranchToAddr(new_pc, BranchType ERET);
```

## Library pseudocode for aarch64/instrs/countop/CountOp

```
enumeration CountOp {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

## Library pseudocode for aarch64/instrs/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =====
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType UXTB;
        when '001' return ExtendType UXTH;
        when '010' return ExtendType UXTW;
        when '011' return ExtendType UXTX;
        when '100' return ExtendType SXTB;
        when '101' return ExtendType SXTH;
        when '110' return ExtendType SXTW;
        when '111' return ExtendType SXTX;
```

## Library pseudocode for aarch64/instrs/extendreg/ExtendReg

```
// ExtendReg()
// =====
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType type, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;

    case type of
        when ExtendType\_SXTB unsigned = FALSE; len = 8;
        when ExtendType\_SXTH unsigned = FALSE; len = 16;
        when ExtendType\_SXTW unsigned = FALSE; len = 32;
        when ExtendType\_SCTX unsigned = FALSE; len = 64;
        when ExtendType\_UXTB unsigned = TRUE; len = 8;
        when ExtendType\_UXTH unsigned = TRUE; len = 16;
        when ExtendType\_UXTW unsigned = TRUE; len = 32;
        when ExtendType\_UCTX unsigned = TRUE; len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

## Library pseudocode for aarch64/instrs/extendreg/ExtendType

```
enumeration ExtendType {ExtendType\_SXTB, ExtendType\_SXTH, ExtendType\_SXTW, ExtendType\_SCTX,
                        ExtendType\_UXTB, ExtendType\_UXTH, ExtendType\_UXTW, ExtendType\_UCTX};
```

## Library pseudocode for aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```
enumeration FPMaxMinOp {FPMaxMinOp\_MAX, FPMaxMinOp\_MIN,
                       FPMaxMinOp\_MAXNUM, FPMaxMinOp\_MINNUM};
```

## Library pseudocode for aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUUnaryOp

```
enumeration FPUUnaryOp {FPUUnaryOp\_ABS, FPUUnaryOp\_MOV,
                       FPUUnaryOp\_NEG, FPUUnaryOp\_SQRT};
```

## Library pseudocode for aarch64/instrs/float/convert/fpconvop/FPConvOp

```
enumeration FPConvOp {FPConvOp\_CVT\_FtoI, FPConvOp\_CVT\_ItoF,
                     FPConvOp\_MOV\_FtoI, FPConvOp\_MOV\_ItoF,
                     FPConvOp\_CVT\_FtoI\_JS
};
```



## Library pseudocode for aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// =====
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
```



```

// DecodeBitMasks()
// =====

// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)
    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
    bits(6) tmask_or, wmask_or;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = if len < 1 then ReservedValue();
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        ReservedValue(len, 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;
    ();

    S = UInt(imms AND levels);
    R = UInt(immr AND levels);
    diff = S - R;    // 6-bit subtract with borrow

    // From a software perspective, the remaining code is equivalent to:
    //   esize = 1 << len;
    //   d = UInt(diff<len-1:0>);
    //   welem = ZeroExtend(Ones(S + 1), esize);
    //   telem = ZeroExtend(Ones(d + 1), esize);
    //   wmask = Replicate(ROR(welem, R));
    //   tmask = Replicate(telem);
    //   return (wmask, tmask);

    // Compute "top mask"
    tmask_and = diff<5:0> OR NOT(levels);
    tmask_or = diff<5:0> AND levels;

    tmask = Ones(64);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
        OR Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
        OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
        OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
        OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
        OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));

```

```

tmask = ((tmask
    AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
    OR Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

// Compute "wraparound mask"
wmask_and = immr OR NOT(levels);
wmask_or = immr AND levels;

wmask = Zeros(64);
wmask = ((wmask
    AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
    OR Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
// optimization of first step:
// wmask = Replicate(wmask_or<0> : '0', 32);
wmask = ((wmask
    AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
    OR Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
wmask = ((wmask
    AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
    OR Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
    AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
    OR Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask
    AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
    OR Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
wmask = ((wmask
    AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
    OR Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

if diff<6> != '0' then // borrow from S - R
    wmask = wmask AND tmask;
else
    wmask = wmask OR tmask;

return (wmask<M-1:0>, tmask<M-1:0>);

```

## Library pseudocode for aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp

```

enumeration MoveWideOp {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};

```

## Library pseudocode for aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// =====
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && immN:imms != '1xxxxxx' then
        return FALSE;
    if sf == '0' && immN:imms != '00xxxxx' then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if S < 16 then
        // ones must not span halfword boundary when rotated
        return (-R MOD 16) <= (15 - S);

    // for MOVN must contain no more than 16 zeros
    if S >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (R MOD 16) <= (S - (width - 15));

    return FALSE;
```

## Library pseudocode for aarch64/instrs/integer/shiftreg/DecodeShift

```
// DecodeShift()
// =====
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00' return ShiftType_LSL;
        when '01' return ShiftType_LSR;
        when '10' return ShiftType_ASR;
        when '11' return ShiftType_ROR;
```

## Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftReg

```
// ShiftReg()
// =====
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType type, integer amount)
    bits(N) result = X[reg];
    case type of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

## Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftType

```
enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

## Library pseudocode for aarch64/instrs/logicalop/LogicalOp

```
enumeration LogicalOp {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

## Library pseudocode for aarch64/instrs/memory/memop/MemAtomicOp

```
enumeration MemAtomicOp {MemAtomicOp_ADD,
                          MemAtomicOp_BIC,
                          MemAtomicOp_EOR,
                          MemAtomicOp_ORR,
                          MemAtomicOp_SMAX,
                          MemAtomicOp_SMIN,
                          MemAtomicOp_UMAX,
                          MemAtomicOp_UMIN,
                          MemAtomicOp_SWP};
```

## Library pseudocode for aarch64/instrs/memory/memop/MemOp

```
enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

## Library pseudocode for aarch64/instrs/memory/prefetch/Prefetch

```
// Prefetch()
// =====

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;           // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;           // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;          // PST: prepare for store
        when '11' return;                          // unallocated hint
    target = UInt(prfop<2:1>);                      // target cache level
    stream = (prfop<0> != '0');                     // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

## Library pseudocode for aarch64/instrs/system/barriers/barrierop/MemBarrierOp

```
enumeration MemBarrierOp { MemBarrierOp_DSB      // Data Synchronization Barrier
                          , MemBarrierOp_DMB      // Data Memory Barrier
                          , MemBarrierOp_ISB      // Instruction Synchronization Barrier
                          , MemBarrierOp_SSBB      // Speculative Synchronization Barrier to VA
                          , MemBarrierOp_PSSBB     // Speculative Synchronization Barrier to PA
                          , MemBarrierOp_SB        // Speculation Barrier
};
```

## Library pseudocode for aarch64/instrs/system/hints/syshintop/SystemHintOp

```
enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI, SystemHintOp_CSDB
};
SystemHintOp_CSDB
};
```

## Library pseudocode for aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFFSet, PSTATEField_DAIFFClr,  
    PSTATEField_PAN, // ARMv8.1  
    PSTATEField_UAO, // ARMv8.2  
    PSTATEField_DIT, // ARMv8.4  
    PSTATEField_SP, PSTATEField_SP  
};  
PSTATEField_SSBS  
};
```

## Library pseudocode for aarch64/instrs/system/sysops/sysop/SysOp

```
// SysOp()
// =====

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
case op1:CRn:CRm:op2 of
  when '000 0111 1000 000' return Sys_AT; // S1E1R
  when '100 0111 1000 000' return Sys_AT; // S1E2R
  when '110 0111 1000 000' return Sys_AT; // S1E3R
  when '000 0111 1000 001' return Sys_AT; // S1E1W
  when '100 0111 1000 001' return Sys_AT; // S1E2W
  when '110 0111 1000 001' return Sys_AT; // S1E3W
  when '000 0111 1000 010' return Sys_AT; // S1E0R
  when '000 0111 1000 011' return Sys_AT; // S1E0W
  when '100 0111 1000 100' return Sys_AT; // S12E1R
  when '100 0111 1000 101' return Sys_AT; // S12E1W
  when '100 0111 1000 110' return Sys_AT; // S12E0R
  when '100 0111 1000 111' return Sys_AT; // S12E0W
  when '011 0111 0100 001' return Sys_DC; // ZVA
  when '000 0111 0110 001' return Sys_DC; // IVAC
  when '000 0111 0110 010' return Sys_DC; // ISW
  when '011 0111 1010 001' return Sys_DC; // CVAC
  when '000 0111 1010 010' return Sys_DC; // CSW
  when '011 0111 1011 001' return Sys_DC; // CVAU
  when '011 0111 1110 001' return Sys_DC; // CIVAC
  when '000 0111 1110 010' return Sys_DC; // CISW
  when '000 0111 0001 000' return Sys_IC; // IALLUIS
  when '000 0111 0101 000' return Sys_IC; // IALLU
  when '011 0111 0101 001' return Sys_IC; // IVAU
  when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
  when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
  when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
  when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
  when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
  when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
  when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
  when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
  when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
  when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
  when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
  when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
  when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
  when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
  when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
  when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
  when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
  when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
  when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
  when '100 1000 0111 000' return Sys_TLBI; // ALLE2
  when '110 1000 0111 000' return Sys_TLBI; // ALLE3
  when '000 1000 0111 001' return Sys_TLBI; // VAE1
  when '100 1000 0111 001' return Sys_TLBI; // VAE2
  when '110 1000 0111 001' return Sys_TLBI; // VAE3
  when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
  when '000 1000 0111 011' return Sys_TLBI; // VAAE1
  when '100 1000 0111 100' return Sys_TLBI; // ALLE1
  when '000 1000 0111 101' return Sys_TLBI; // VALE1
  when '100 1000 0111 101' return Sys_TLBI; // VALE2
  when '110 1000 0111 101' return Sys_TLBI; // VALE3
  when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
  when '000 1000 0111 111' return Sys_TLBI; // VAALE1
return Sys_SYS;
```

## Library pseudocode for aarch64/instrs/system/sysops/sysop/SystemOp

```
enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```



## Library pseudocode for aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```
enumeration VBitOp      {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

## Library pseudocode for aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```
enumeration CompareOp   {CompareOp_GT, CompareOp_GE, CompareOp_EQ,  
                          CompareOp_LE, CompareOp_LT};
```

## Library pseudocode for aarch64/instrs/vector/logical/immediateop/ImmediateOp

```
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,  
                          ImmediateOp_ORR, ImmediateOp_BIC};
```

## Library pseudocode for aarch64/instrs/vector/reduce/reduceop/Reduce

```
// Reduce()  
// =====  
  
bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)  
    integer half;  
    bits(esize) hi;  
    bits(esize) lo;  
    bits(esize) result;  
  
    if N == esize then  
        return input<esize-1:0>;  
  
    half = N DIV 2;  
    hi = Reduce(op, input<N-1:half>, esize);  
    lo = Reduce(op, input<half-1:0>, esize);  
  
    case op of  
        when ReduceOp\_FMINNUM  
            result = FPMinNum(lo, hi, FPCR);  
        when ReduceOp\_FMAXNUM  
            result = FPMaxNum(lo, hi, FPCR);  
        when ReduceOp\_FMIN  
            result = FPMin(lo, hi, FPCR);  
        when ReduceOp\_FMAX  
            result = FPMax(lo, hi, FPCR);  
        when ReduceOp\_FADD  
            result = FPAdd(lo, hi, FPCR);  
        when ReduceOp\_ADD  
            result = lo + hi;  
  
    return result;
```

## Library pseudocode for aarch64/instrs/vector/reduce/reduceop/ReduceOp

```
enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,  
                      ReduceOp_FMIN, ReduceOp_FMAX,  
                      ReduceOp_FADD, ReduceOp_ADD};
```

## Library pseudocode for aarch64/translation/attrs/AArch64.InstructionDevice

```
// AArch64.InstructionDevice()
// =====
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch64.InstructionDevice(AddressDescriptor addrdesc, bits(64) vaddress,
                                           bits(52) ipaddress, integer level,
                                           AccType acctype, boolean iswrite, boolean secondstage,
                                           boolean s2fslwalk)

c = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
assert c IN {Constraint\_NONE, Constraint\_FAULT};

if c == Constraint\_FAULT then
    addrdesc.fault = AArch64.PermissionFault(ipaddress, bit UNKNOWN, level, acctype, iswrite,
                                             secondstage, s2fslwalk);
else
    addrdesc.memattrs.type = MemType\_Normal;
    addrdesc.memattrs.inner.attrs = MemAttr\_NC;
    addrdesc.memattrs.inner.hints = MemHint\_No;
    addrdesc.memattrs.outer = addrdesc.memattrs.inner;
    addrdesc.memattrs.tagged = FALSE;
    addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

return addrdesc;
```

## Library pseudocode for aarch64/translation/attrs/AArch64.S1AttrDecode

```
// AArch64.S1AttrDecode()
// =====
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch64.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;

    mair = MAIR[];
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    memattrs.tagged = FALSE;
    if ((attrfield<7:4> != '0000' && attrfield<7:4> != '1111' && attrfield<3:0> == '0000') ||
if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable\_RESMAIR);
    if !
if attrfield<7:4> == '0000' then // Device
    memattrs.type = HaveMTEExt() && attrfield<7:4> == '1111' && attrfield<3:0> == '0000' then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable\_RESMAIR);

    if attrfield<7:4> == '0000' then // Device
        memattrs.type = MemType\_Device;
        case attrfield<3:0> of
            when '0000' memattrs.device = DeviceType\_nGnRnE;
            when '0100' memattrs.device = DeviceType\_nGnRE;
            when '1000' memattrs.device = DeviceType\_nGRE;
            when '1100' memattrs.device = DeviceType\_GRE;
            otherwise Unreachable(); // Reserved, handled above

    elsif attrfield<3:0> != '0000' then // Normal
        memattrs.type = MemType\_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    elsif HaveMTEExt() && attrfield == '11110000' then // Tagged, Normal
        memattrs.tagged = TRUE;
        memattrs.type = MemType\_Normal;
        memattrs.outer.attrs = MemAttr\_WB;
        memattrs.inner.attrs = MemAttr\_WB;
        memattrs.outer.hints = MemHint\_RWA;
        memattrs.inner.hints = MemHint\_RWA;
(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';

    else
        Unreachable(); // Reserved, handled above

    return MemAttrDefaults(memattrs);
```

## Library pseudocode for aarch64/translation/attrs/AArch64.TranslateAddressS1Off

```
// AArch64.TranslateAddressS1Off()
// =====
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch64.TranslateAddressS1Off(bits(64) vaddress, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    Top = AddrTop(vaddress, (acctype == AccType\_IFETCH), PSTATE.EL);
    if !IsZero(vaddress<Top:PAMax()>) then
        level = 0;
        ipaddress = bits(52) UNKNOWN;
        secondstage = FALSE;
        s2fslwalk = FALSE;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, bit UNKNOWN, level, acctype,
                                                         iswrite, secondstage, s2fslwalk);

        return result;

    default_cacheable = (HasS2Translation() && HCR_EL2.DC == '1');

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.type = MemType\_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr\_WB; // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint\_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
        result.addrdesc.memattrs.tagged = HCR_EL2.DCT == '1';
    elseif acctype != AccType\_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.type = MemType\_Device;
        result.addrdesc.memattrs.device = DeviceType\_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
        result.addrdesc.memattrs.tagged = FALSE;
    else
        // Instruction cacheability controlled by SCTLR_ELx.I
        cacheable = SCTLR[][I] == '1';
        result.addrdesc.memattrs.type = MemType\_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr\_WT;
            result.addrdesc.memattrs.inner.hints = MemHint\_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr\_NC;
            result.addrdesc.memattrs.inner.hints = MemHint\_No;
        result.addrdesc.memattrs.shareable = TRUE;
        result.addrdesc.memattrs.outershareable = TRUE;
        result.addrdesc.memattrs.tagged = FALSE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.address = vaddress<51:0>;
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch64.NoFault();
    return result;
```

## Library pseudocode for aarch64/translation/checks/AArch64.AccessIsPrivileged

```
// AArch64.AccessIsPrivileged()
// =====

boolean AArch64.AccessIsPrivileged(AccType acctype)

    el = el = if () && acctype == AccType_NV2REGISTER then EL2AArch64.AccessUsesELHaveNV2Ext(acctype)
else PSTATE.EL;
    if el == EL0 then
        ispriv = FALSE;
    elsif el == EL3 then
        ispriv = TRUE;
    elsif el == EL2 && (!IsInHost() || HCR_EL2.TGE == '0') then
        ispriv = TRUE;
    elsif HaveUAOExt() && PSTATE.UAO == '1' then
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType_UNPRIV);

    return ispriv;
```



```

// AArch64.AccessUsesEL()
// =====
// Returns the Exception Level of the regime that will manage the translation for a given access type.
// AArch64.CheckPermission()
// =====
// Function used for permission checking from AArch64 stage 1 translations

bits(2) FaultRecord AArch64.AccessUsesEL(AArch64.CheckPermission(Permissions perms, bits(64) vaddress, i
                                bit NS, AccType acctype)
    if acctype == acctype, boolean iswrite)
    assert ! AccType_UNPRIVILELUsingAArch32 then
        return( S1TranslationRegime());

    wxn = SCTLR[].WXN == '1';

    if (PSTATE.EL == EL0;
    elseif acctype == || IsInHost() ||
        (PSTATE.EL == EL1 && !HaveNV2Ext()) ||
        (PSTATE.EL == EL1 && HaveNV2Ext() && (acctype != AccType_NV2REGISTER then
            return || ! ELIsInHost(EL2))) then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';

        ispriv = AArch64.AccessIsPrivileged(acctype);

        pan = if HavePANExt() then PSTATE.PAN else '0';
        if (EL2Enabled() && ((PSTATE.EL == EL1 && HaveNVExt() && HCR_EL2.<NV, NV1> == '11') ||
            (HaveNV2Ext() && acctype == AccType_NV2REGISTER && HCR_EL2.NV2 == '1'))) then
            pan = '0';
        if (pan == '1' && user_r && ispriv &&
            !(acctype IN {AccType_DC, AccType_AT, AccType_IFETCH}) ||
            (acctype == AccType_AT && AArch64.ExecutingATSlxPInstr())) then
            priv_r = FALSE;
            priv_w = FALSE;

        user_xn = perms.xn == '1' || (user_w && wxn);
        priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2 or EL3
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

        // Restriction on Secure instruction fetch
        if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
            xn = TRUE;

        if acctype == AccType_IFETCH then
            fail = xn;
            failedread = TRUE;
        elseif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW } then
            fail = !r || !w;
            failedread = !r;
        elseif iswrite then
            fail = !w;
            failedread = FALSE;
        elseif acctype == AccType_DC && ispriv then
            // DC maintenance instructions operating by VA, cannot fault from stage 1 translation,
            // other than DC IVAC, which requires write permission, and operations executed at EL0,
            // which require read permission.
            fail = FALSE;
        else
            fail = !r;

```

```

failedread = TRUE;

if fail then
    secondstage = FALSE;
    s2fslwalk = FALSE;
    ipaddress = bits(52) UNKNOWN;
    return AArch64.PermissionFault(ipaddress, bit UNKNOWN, level, acctype,
                                   !failedread, secondstage, s2fslwalk);
else
    return AArch64.NoFault;
else
    return PSTATE.EL; ();

```





```

// AArch64.CheckPermission()
// =====
// Function used for permission checking from AArch64 stage 1 translations
// AArch64.CheckS2Permission()
// =====
// Function used for permission checking from AArch64 stage 2 translations

FaultRecord AArch64.CheckPermission(AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, int
                                bit NS, perms, bits(64) vaddress, bits(52) ipaddress,
                                integer level, AccType acctype, boolean iswrite)
    assert !acctype, boolean iswrite, bit NS,
                                boolean s2fslwalk, boolean hwupdatewalk)

    assert ELUsingAArch32IsSecureEL2Enabled(() || (S1TranslationRegimeHaveEL()));

    wxn = ( SCTLREL2[].WXN == '1';

    if (PSTATE.EL ==) &&! EL0IsSecure ||() &&!
        IsInHostELUsingAArch32() ||
        (PSTATE.EL == ( EL1 && !HaveNV2Ext() ) ||
        (PSTATE.EL == EL1 && HaveNV2Ext() && (acctype != AccType_NV2REGISTER || !ELIsInHost(EL2)))) the
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';

        ispriv =) ) && AArch64.AccessIsPrivilegedHasS2Translation(acctype);
(});

    pan = if r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    if HavePANExtHaveExtendedExecuteNeverExt() then PSTATE.PAN else '0';
        if () then
            case perms.xn:perms.xxn of
                when '00' xn = FALSE;
                when '01' xn = PSTATE.EL == EL2Enabled() && ((PSTATE.EL == EL1 &&
                when '10' xn = TRUE;
                when '11' xn = PSTATE.EL == HaveNVExtEL0() && HCR_EL2.<NV, NV1> == '11') ||
            (};
        else
            xn = perms.xn == '1';
            // Stage 1 walk is checked as a read, regardless of the original type
            if acctype == HaveNV2Ext() && acctype == AccType_NV2REGISTER && HCR_EL2.NV2 == '1')) then
                pan = '0';
                is_ldst = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
                is_atslxp = (acctype == && !s2fslwalk then
                fail = xn;
                failedread = TRUE;
            elseif (acctype IN { AccType_AT && AArch64.ExecutingATSlxPInstr());
                if pan == '1' && user_r && ispriv && (is_ldst || is_atslxp) then
                    priv_r = FALSE;
                    priv_w = FALSE;

                    user_xn = perms.xn == '1' || (user_w && wxn);
                    priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;

                    if ispriv then
                        (r, w, xn) = (priv_r, priv_w, priv_xn);
                    else
                        (r, w, xn) = (user_r, user_w, user_xn);
                else
                    // Access from EL2 or EL3
                    r = TRUE;
                    w = perms.ap<2> == '0';
                    xn = perms.xn == '1' || (w && wxn);

                    // Restriction on Secure instruction fetch
                    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
                        xn = TRUE;

```

```

    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW } then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    elsif acctype == AccType_DC && PSTATE.EL != EL0 then
        // DC maintenance instructions operating by VA, cannot fault from stage 1 translation,
        // other than DC IVAC, which requires write permission, and operations executed at EL0,
        // which require read permission.
        fail = FALSE;
    }) && !s2fslwalk then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite && !s2fslwalk then
        fail = !w;
        failedread = FALSE;
    elsif hwupdatewalk then
        fail = !w;
        failedread = !iswrite;
    else
        fail = !r;
        failedread = TRUE;
        failedread = !iswrite;

    if fail then
        secondstage = FALSE;
        s2fslwalk = FALSE;
        ipaddress = bits(52) UNKNOWN;
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch64.PermissionFault(ipaddress, bit UNKNOWN, level, acctype,
        (ipaddress, NS, level, acctype,
                                !failedread, secondstage, s2fslwalk);
    else
        return AArch64.NoFault();

```

## Library pseudocode for aarch64/translation/checks/AArch64.CheckS2Permission

```
// AArch64.CheckS2Permission()
// =====
// Function used for permission checking from AArch64 stage 2 translations

FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, bits(52) ipaddress,
                                       integer level, AccType acctype, boolean iswrite, bit NS,
                                       boolean s2fslwalk, boolean hwupdatewalk)

    assert IsSecureEL2Enabled() || ( HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) ) && HasS2Trans

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    if HaveExtendedExecuteNeverExt() then
        case perms.xn:perms.xxn of
            when '00' xn = FALSE;
            when '01' xn = PSTATE.EL == EL1;
            when '10' xn = TRUE;
            when '11' xn = PSTATE.EL == EL0;
        else
            xn = perms.xn == '1';
        // Stage 1 walk is checked as a read, regardless of the original type
        if acctype == AccType IFETCH && !s2fslwalk then
            fail = xn;
            failedread = TRUE;
        elsif (acctype IN { AccType ATOMICRW, AccType ORDEREDRW, AccType ORDEREDATOMICRW }) && !s2fslwalk then
            fail = !r || !w;
            failedread = !r;
        elsif iswrite && !s2fslwalk then
            fail = !w;
            failedread = FALSE;
        elsif acctype == AccType DC && PSTATE.EL != EL0 && !s2fslwalk then
            // DC maintenance instructions operating by VA, with the exception of DC IVAC, do
            // not generate Permission faults from stage 2 translation, other than when
            // performing a stage 1 translation table walk.
            fail = FALSE;
        elsif hwupdatewalk then
            fail = !w;
            failedread = !iswrite;
        else
            fail = !r;
            failedread = !iswrite;

        if fail then
            domain = bits(4) UNKNOWN;
            secondstage = TRUE;
            return AArch64.PermissionFault(ipaddress, NS, level, acctype,
                                           !failedread, secondstage, s2fslwalk);
        else
            return AArch64.NoFault();
```

## Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, AccType acctype, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    match = FALSE;

    for i = 0 to UInt(ID_AA64DFR0_EL1.BRPs)
        match_i = AArch64.BreakpointMatch(i, vaddress, acctype, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt\_Breakpoint;
        Halt(reason);
    elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        acctype = AccType\_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

## Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch64.NoFault();

    d_side = (acctype != AccType\_IFETCH);
    generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch64.CheckBreakpoint(vaddress, acctype, size);

    return fault;
```

## Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                     boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = AArch64.AccessIsPrivileged(acctype);

    for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
        match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        if acctype != AccType\_NONFAULT && acctype != AccType\_CNOTFIRST then
            reason = DebugHalt\_Watchpoint;
            Halt(reason);
        else
            // Fault will be reported and cancelled
            return AArch64.DebugFault(acctype, iswrite);
    elseif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

## Library pseudocode for aarch64/translation/faults/AArch64.AccessFlagFault

```
// AArch64.AccessFlagFault()
// =====

FaultRecord AArch64.AccessFlagFault(bits(52) ipaddress, bit NS, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fslwalk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault\_AccessFlag, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch64/translation/faults/AArch64.AddressSizeFault

```
// AArch64.AddressSizeFault()
// =====

FaultRecord AArch64.AddressSizeFault(bits(52) ipaddress, bit NS, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fslwalk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault\_AddressSize, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch64/translation/faults/AArch64.AlignmentFault

```
// AArch64.AlignmentFault()
// =====

FaultRecord AArch64.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fslwalk = boolean UNKNOWN;

    return AArch64.CreateFaultRecord(Fault\_Alignment, ipaddress, bit UNKNOWN, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch64/translation/faults/AArch64.AsynchExternalAbort

```
// AArch64.AsynchExternalAbort()
// =====
// Wrapper function for asynchronous external aborts

FaultRecord AArch64.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

    type = if parity then Fault\_AsyncParity else Fault\_AsyncExternal;
    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType\_NORMAL;
    iswrite = boolean UNKNOWN;
    secondstage = FALSE;
    s2fslwalk = FALSE;

    return AArch64.CreateFaultRecord(type, ipaddress, bit UNKNOWN, level, acctype, iswrite, extflag,
                                     errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch64/translation/faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// =====

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)

    ipaddress = bits(52) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fslwalk = FALSE;

    return AArch64.CreateFaultRecord(Fault\_Debug, ipaddress, bit UNKNOWN, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch64/translation/faults/AArch64.NoFault

```
// AArch64.NoFault()
// =====

FaultRecord AArch64.NoFault()

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fslwalk = FALSE;

    return AArch64.CreateFaultRecord(Fault None, ipaddress, bit UNKNOWN, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch64/translation/faults/AArch64.PermissionFault

```
// AArch64.PermissionFault()
// =====

FaultRecord AArch64.PermissionFault(bits(52) ipaddress, bit NS, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fslwalk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault Permission, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fslwalk);
```

## Library pseudocode for aarch64/translation/faults/AArch64.TranslationFault

```
// AArch64.TranslationFault()
// =====

FaultRecord AArch64.TranslationFault(bits(52) ipaddress, bit NS, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fslwalk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault Translation, ipaddress, NS, level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fslwalk);
```





```

// AArch64.CheckAndUpdateDescriptor()
// =====
// Check and update translation table descriptor if hardware update is configured

FaultRecord AArch64.CheckAndUpdateDescriptor(DescriptorUpdate result, FaultRecord fault,
                                              boolean secondstage, bits(64) vaddress, AccType acctype,
                                              boolean iswrite, boolean s2fslwalk, boolean hwupdatewalk)

// Check if access flag can be updated
// Address translation instructions are permitted to update AF but not required
if result.AF then
    if fault.type == Fault_None then
        hw_update_AF = TRUE;
    elsif ConstrainUnpredictable(Unpredictable_AFUPDATE) == Constraint_TRUE then
        hw_update_AF = TRUE;
    else
        hw_update_AF = FALSE;

if result.AP && fault.type == Fault_None then
    write_perm_req = (iswrite || acctype IN {AccType_ATOMICRW, AccType_ORDEREDRW, }) && !s2fslwalk;
    hw_update_AP = (write_perm_req && !(acctype IN { AccType_ORDEREDATOMICRW } ) && !s2fslwalk;
    hw_update_AP = (write_perm_req && !(acctype IN {AccType_AT, AccType_DC, })) || hwupdatewalk;
else
    hw_update_AP = FALSE;

if hw_update_AF || hw_update_AP then
    if secondstage || ! AccType_DC_UNPRIV))) || hwupdatewalk;
else
    hw_update_AP = FALSE;

if hw_update_AF || hw_update_AP then
    if secondstage || !HasS2Translation() then
        descaddr2 = result.descaddr;
    else
        hwupdatewalk = TRUE;
        descaddr2 = AArch64.SecondStageWalk(result.descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
        if IsFault(descaddr2) then
            return descaddr2.fault;

    accdesc = CreateAccessDescriptor(----- accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
    desc = _Mem[descaddr2, 8, accdesc];
    el = AArch64.AccessUsesEL(acctype);
    case el of
        when EL3
            reversedescriptors = SCTLR_EL3.EE == '1';
        when EL2
            reversedescriptors = SCTLR_EL2.EE == '1';
        otherwise
            reversedescriptors = SCTLR_EL1.EE == '1';
    if reversedescriptors then
        desc = BigEndianReverse(desc);

    if hw_update_AF then
        desc<10> = '1';
    if hw_update_AP then
        desc<7> = (if secondstage then '1' else '0');

    _Mem[descaddr2,8,accdesc] = if reversedescriptors then BigEndianReverse(desc) else desc;
);
----- desc = _Mem[descaddr2, 8, accdesc];
----- if hw_update_AF then
-----     desc<10> = '1';
----- if hw_update_AP then
-----     desc<7> = (if secondstage then '1' else '0');
----- _Mem[descaddr2,8,accdesc] = desc;

return fault;

```

## Library pseudocode for aarch64/translation/translation/AArch64.FirstStageTranslate

```
// AArch64.FirstStageTranslate()
// =====
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.FirstStageTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                             boolean wasaligned, integer size)

if HaveNV2Ext() && acctype == AccType\_NV2REGISTER then
    s1_enabled = SCTLR_EL2.M == '1';
elseif HasS2Translation() then
    s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';
else
    s1_enabled = SCTLR[].M == '1';

ipaddress = bits(52) UNKNOWN;
secondstage = FALSE;
s2fslwalk = FALSE;

if s1_enabled then                                     // First stage enabled
    S1 = AArch64.TranslationTableWalk(ipaddress, '1', vaddress, acctype, iswrite, secondstage,
                                       s2fslwalk, size);
    permissioncheck = TRUE;
    if acctype == else
    S1 = AccType\_IFETCH then
        InGuardedPage = S1.GP == '1';                // Global state updated on instruction fetch that denotes
                                                         // if the fetched instruction is from a guarded page.
    else
        S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
    permissioncheck = FALSE;
    if UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() then
        if S1.addrdesc.memattrs.type == MemType\_Device && S1.addrdesc.memattrs.device != DeviceType
            nTLSMD = if S1TranslationRegime() == EL2 then SCTLR_EL2.nTLSMD else SCTLR_EL1.nTLSMD;
            if nTLSMD == '0' then
                S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

// Check for unaligned data accesses to Device memory
if ((!wasaligned && acctype != AccType\_IFETCH) || (acctype == AccType\_DCZVA)
    && S1.addrdesc.memattrs.type == MemType\_Device && !IsFault(S1.addrdesc) then
    S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
if !IsFault(S1.addrdesc) && permissioncheck then
    S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level,
                                              S1.addrdesc.paddress.NS,
                                              acctype, iswrite);

// Check for instruction fetches from Device memory not marked as execute-never. If there has
// not been a Permission Fault then the memory is not marked execute-never.
if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType\_Device &&
    acctype == AccType\_IFETCH) then
    S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                           acctype, iswrite,
                                           secondstage, s2fslwalk);

// Check and update translation table descriptor if required
hwupdatewalk = FALSE;
s2fslwalk = FALSE;
S1.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S1.descupdate, S1.addrdesc.fault,
                                                    secondstage, vaddress, acctype,
                                                    iswrite, s2fslwalk, hwupdatewalk);

return S1.addrdesc;
```

## Library pseudocode for aarch64/translation/translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =====
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch64.FullTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                         boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch64.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(S1) && !(HaveNV2Ext()) && acctype == AccType\_NV2REGISTER && HasS2Translation() then
        s2fslwalk = FALSE;
        hwupdatewalk = FALSE;
        result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fslwalk,
                                                size, hwupdatewalk);
    else
        result = S1;

    return result;
```

## Library pseudocode for aarch64/translation/translation/AArch64.SecondStageTranslate

```
// AArch64.SecondStageTranslate()
// =====
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.SecondStageTranslate(AddressDescriptor S1, bits(64) vaddress,
                                                AccType acctype, boolean iswrite, boolean wasaligned,
                                                boolean s2fslwalk, integer size, boolean hwupdatewalk)

    assert HasS2Translation();

    s2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';
    secondstage = TRUE;

    if s2_enabled then // Second stage enabled
        ipaddress = S1.paddress.address<51:0>;
        NS = S1.paddress.NS;
        S2 = AArch64.TranslationTableWalk(ipaddress, NS, vaddress, acctype, iswrite, secondstage,
                                           s2fslwalk, size);

        // Check for unaligned data accesses to Device memory
        if ((!wasaligned && acctype != AccType IFETCH) || (acctype == AccType DCZVA))
            && S2.addrdesc.memattrs.type == MemType Device && !IsFault(S2.addrdesc) then
                S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

        // Check for permissions on Stage2 translations
        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                         acctype, iswrite, NS, s2fslwalk, hwupdatewalk);

        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!s2fslwalk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType Device &&
            acctype == AccType IFETCH) then
            S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    acctype, iswrite,
                                                    secondstage, s2fslwalk);

        // Check for protected table walk
        if (s2fslwalk && !IsFault(S2.addrdesc) && HCR_EL2.PTW == '1' &&
            S2.addrdesc.memattrs.type == MemType Device) then
            S2.addrdesc.fault = AArch64.PermissionFault(ipaddress, S1.paddress.NS, S2.level, acctype,
                                                         iswrite, secondstage, s2fslwalk);

        // Check and update translation table descriptor if required
        S2.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S2.descupdate, S2.addrdesc.fault,
                                                            secondstage, vaddress, acctype,
                                                            iswrite, s2fslwalk, hwupdatewalk);

        result = CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

    return result;
```

## Library pseudocode for aarch64/translation/translation/AArch64.SecondStageWalk

```
// AArch64.SecondStageWalk()
// =====
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype,
                                           boolean iswrite, integer size, boolean hwupdatewalk)

    assert HasS2Translation();

    s2fslwalk = TRUE;
    wasaligned = TRUE;
    return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fslwalk,
                                         size, hwupdatewalk);
```

## Library pseudocode for aarch64/translation/translation/AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)

    result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType PTW, AccType IC, AccType AT}) && !IsFault(result) then
        result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
```



```

// AArch64.TranslationTableWalk()
// =====
// Returns a result of a translation table walk
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch64.TranslationTableWalk(bits(52) ipaddress, bit s1_nonsecure, bits(64) vaddress,
                                         AccType acctype, boolean iswrite, boolean secondstage,
                                         boolean s2fslwalk, integer size)

if !secondstage then
    assert !ELUsingAArch32(S1TranslationRegime());
else
    assert IsSecureEL2Enabled() || ( HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) ) && HasS2TLB

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(64) inputaddr;          // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2

    descaddr.memattrs.type = MemType_Normal;

    // Derived parameters for the page table walk:
    // grainsize = Log2(Size of Table)          - Size of Table is 4KB, 16KB or 64KB in AArch64
    // stride = Log2(Address per Level)         - Bits of address consumed at each level
    // firstblocklevel = First level where a block entry is allowed
    // ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTCR_EL2.PS
    // inputsize = Log2(Size of Input Address) - Input Address size in bits
    // level = Level to start walk from
    // This means that the number of levels after start level = 3-level

    if !secondstage then
        // First stage translation
        inputaddr = ZeroExtend(vaddress);
        el = el = if AArch64.AccessUsesELHaveNV2Ext(acctype);
        top = () && acctype == AccType_NV2REGISTER then EL2 else PSTATE_EL2;
        top = AddrTop(inputaddr, (acctype == AccType_IFETCH), el);
        if el == EL3 then
            largegrain = TCR_EL3.TG0 == '01';
            midgrain = TCR_EL3.TG0 == '10';
            inputsize = 64 - UInt(TCR_EL3.TOSZ);
            inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
            inputsize_min = 64 - (if !() && largegrain then 52 else 48);
            if inputsize > inputsize_max then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            ps = TCR_EL3.PS;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr < top);
            disabled = FALSE;
            baseregister = TTBR0_EL3;
            descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGNO, TCR_EL3.IRGNO, secondstage);
            reversedescriptors = SCTLRL_EL3.EE == '1';
            lookupsecure = TRUE;
            singlepriv = TRUE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL3.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL3.HD == '1';
            hierattrdisabled = AArch64.HaveHPDEExt() && TCR_EL3.HPD == '1';
        elseif ELIsInHost(el) then
            if inputaddr < top == '0' then
                largegrain = TCR_EL2.TG0 == '01';
                midgrain = TCR_EL2.TG0 == '10';
                inputsize = 64 - UInt(TCR_EL2.TOSZ);
                inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;

```



```

        inputsize_min = 64 - (if !() && largegrain then 52 else 48);
        if inputsize > inputsize_max then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr < TCR_EL2.EPD0);
            disabled = TCR_EL2.EPD0 == '1' || (PSTATE.EL == 2 && disabled = TCR_EL2.EPD0 == '1');
            baseregister = TTBR0_EL2;
            descaddr.memattrs = EL0 && HaveE0PDEExt() && TCR_EL2.E0PD0 == '1';
            baseregister = TTBR0_EL2;
            descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGNO, TCR_EL2.IRGNO, secondstage);
            hierattrsdissabled = AArch64.HaveHPDEExt() && TCR_EL2.HPD0 == '1';
        else
            inputsize = 64 - UInt(TCR_EL2.T1SZ);
            largegrain = TCR_EL2.TG1 == '11'; // TG1 and TG0 encodings differ
            midgrain = TCR_EL2.TG1 == '01';
            inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
            inputsize_min = 64 - (if !() && largegrain then 52 else 48);
            if inputsize > inputsize_max then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputaddr < TCR_EL2.EPD1);
            disabled = TCR_EL2.EPD1 == '1' || (PSTATE.EL == 2 && disabled = TCR_EL2.EPD1 == '1');
            baseregister = TTBR1_EL2;
            descaddr.memattrs = EL0 && HaveE0PDEExt() && TCR_EL2.E0PD1 == '1';
            baseregister = TTBR1_EL2;
            descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH1, TCR_EL2.ORGNO, TCR_EL2.IRGNO, secondstage);
            hierattrsdissabled = AArch64.HaveHPDEExt() && TCR_EL2.HPD1 == '1';
        ps = TCR_EL2.IPS;
        reversedescriptors = SCTLR_EL2.EE == '1';
        lookupsecure = if IsSecureEL2Enabled() then IsSecure() else FALSE;
        singlepriv = FALSE;
        update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
    elseif el == EL2 then
        inputsize = 64 - UInt(TCR_EL2.T0SZ);
        largegrain = TCR_EL2.TG0 == '01';
        midgrain = TCR_EL2.TG0 == '10';
        inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
        inputsize_min = 64 - (if !() && largegrain then 52 else 48);
        if inputsize > inputsize_max then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_max;
        inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
        if inputsize < inputsize_min then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_min;
        ps = TCR_EL2.PS;
        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr < TCR_EL2.EPD0);
        disabled = FALSE;
        baseregister = TTBR0_EL2;
        descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGNO, TCR_EL2.IRGNO, secondstage);
        reversedescriptors = SCTLR_EL2.EE == '1';
        lookupsecure = if IsSecureEL2Enabled() then IsSecure() else FALSE;
        singlepriv = TRUE;
        update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';

```

```

        hierattrsdissabled = AArch64.HaveHPDExt() && TCR_EL2.HPD == '1';
    else
        if inputaddr<top> == '0' then
            inputsize = 64 - UInt(TCR_EL1.T0SZ);
            largegrain = TCR_EL1.TG0 == '01';
            midgrain = TCR_EL1.TG0 == '10';
            inputsize_max = if Have52BitVAExt() && largegrain then
            inputsize_min = 64 - (if !() && largegrain then 52 else 48);
            if inputsize > inputsize_max then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr
            disabled = TCR_EL1.EPD0 == '1' || (PSTATE.EL == disabled = TCR_EL1.EPD0
            disabled = disabled || (el == EL0 &&& acctype == HaveE0PDEExt() && TCR_EL1.E0PD0 == '1'
            disabled = disabled || (el == EL0 && acctype == AccType_NONFAULT && TCR_EL1.NFD0 == '1'
            baseregister = TTBR0_EL1;
            descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORGNO, TCR_EL1.IRGNO, secondsta
            hierattrsdissabled = AArch64.HaveHPDExt() && TCR_EL1.HPD0 == '1';
        else
            inputsize = 64 - UInt(TCR_EL1.T1SZ);
            largegrain = TCR_EL1.TG1 == '11'; // TG1 and TG0 encodings differ
            midgrain = TCR_EL1.TG1 == '01';
            inputsize_max = if Have52BitVAExt() && largegrain then
            inputsize_min = 64 - (if !() && largegrain then 52 else 48);
            if inputsize > inputsize_max then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputaddr
            disabled = TCR_EL1.EPD1 == '1' || (PSTATE.EL == disabled = TCR_EL1.EPD1
            disabled = disabled || (el == EL0 &&& acctype == HaveE0PDEExt() && TCR_EL1.E0PD1 == '1'
            disabled = disabled || (el == EL0 && acctype == AccType_NONFAULT && TCR_EL1.NFD1 == '1'
            baseregister = TTBR1_EL1;
            descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGNO, TCR_EL1.IRGNO, secondsta
            hierattrsdissabled = AArch64.HaveHPDExt() && TCR_EL1.HPD1 == '1';
        ps = TCR_EL1.IPS;
        reversedescriptors = SCTL_EL1.EE == '1';
        lookupsecure = IsSecure();
        singlepriv = FALSE;
        update_AF = HaveAccessFlagUpdateExt() && TCR_EL1.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL1.HD == '1';
    if largegrain then
        grainsize = 16; // Log2(64KB page size)
        firstblocklevel = (if Have52BitPAExt() then 1 else 2); // Largest block is 4TB (2^42 b
        // and 512MB (2^29 bytes) other
    elsif midgrain then
        grainsize = 14; // Log2(16KB page size)
        firstblocklevel = 2; // Largest block is 32MB (2^25
    else // Small grain
        grainsize = 12; // Log2(4KB page size)
        firstblocklevel = 1; // Largest block is 1GB (2^30 b
    stride = grainsize - 3; // Log2(page size / 8 bytes)
    // The starting level is the number of strides needed to consume the input address
    level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));

else
    // Second stage translation
    inputaddr = ZeroExtend(ipaddress);
    // Stage 2 translation table walk for the Secure EL2 translation regime

```

```

if IsSecureEL2Enabled() && IsSecure() then

    // Stage 2 translation walk is in the Non-secure IPA space or the Secure IPA space
    t0size = if s1_nonsecure == '1' then VTCR_EL2.T0SZ else VSTCR_EL2.T0SZ;
    tg0     = if s1_nonsecure == '1' then VTCR_EL2.TG0 else VSTCR_EL2.TG0;

    // Stage 2 translation table walk is to the Non-secure PA space or to the Secure PA space
    nswalk = if s1_nonsecure == '1' then VTCR_EL2.NSW else VSTCR_EL2.SW;

    // Stage 2 translation accesses the Non-secure PA space or the Secure PA space
    if nswalk == '1' then
        nsaccess = '1'; // When walk is non-secure, access is non-secure
    else
        if s1_nonsecure == '0' then
            nsaccess = VSTCR_EL2.SA; // When walk is secure and in the Non-secure PA space
        else
            if VSTCR_EL2.SW == '1' || VSTCR_EL2.SA == '1' then nsaccess = '1'; // When walk is secure and in the Secure PA space
            else nsaccess = VTCR_EL2.NSA; // When walk is secure and in the Non-secure PA space
    else
        t0size = VTCR_EL2.T0SZ;
        tg0     = VTCR_EL2.TG0;
        nsaccess = '1';

    inputsize = 64 - UInt(t0size);
    largegrain = tg0 == '01';
    midgrain   = tg0 == '10';

    inputsize_max = if inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48;
    inputsize_min = 64 - (if !() && largegrain then 52 else 48);
    if inputsize > inputsize_max then
        c = ConstrainUnpredictable(Unpredictable RESTnSZ);
        assert c IN {Constraint FORCE, Constraint FAULT};
        if c == Constraint FORCE then inputsize = inputsize_max;
    inputsize_min = 64 - (if !HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);
    if inputsize < inputsize_min then
        c = ConstrainUnpredictable(Unpredictable RESTnSZ);
        assert c IN {Constraint FORCE, Constraint FAULT};
        if c == Constraint FORCE then inputsize = inputsize_min;
    ps = VTCR_EL2.PS;
    basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<63:inputsize);
    disabled = FALSE;
    descaddr.memattrs = WalkAttrDecode(VTCR_EL2.IRGN0, VTCR_EL2.ORGNO, VTCR_EL2.SH0, secondstage);
    reversedescriptors = SCTLR_EL2.EE == '1';
    singlepriv = TRUE;
    update_AF = HaveAccessFlagUpdateExt() && VTCR_EL2.HA == '1';
    update_AP = HaveDirtyBitModifierExt() && update_AF && VTCR_EL2.HD == '1';

    lookupsecure = if IsSecureEL2Enabled() then s1_nonsecure == '0' else FALSE;
    // Stage2 translation table walk is to secure PA space or to Non-secure PA space
    baseregister = if lookupsecure then VSTTBR_EL2 else VTTBR_EL2;
    startlevel   = if lookupsecure then UInt(VSTCR_EL2.SL0) else UInt(VTCR_EL2.SL0);
    if largegrain then
        grainsize = 16; // Log2(64KB page size)
        level = 3 - startlevel;
        firstblocklevel = (if Have52BitPAExt() then 1 else 2); // Largest block is 4TB (2^42 bytes)
        // and 512MB (2^29 bytes) otherwise
    elseif midgrain then
        grainsize = 14; // Log2(16KB page size)
        level = 3 - startlevel;
        firstblocklevel = 2; // Largest block is 32MB (2^25 bytes)
    else // Small grain
        grainsize = 12; // Log2(4KB page size)
        if HaveSmallPageTblExt() && startlevel == 3 then
            level = startlevel; // Startlevel 3 (VTCR_EL2.SL0 or VSTCR_EL2.SL0)
        else
            level = 2 - startlevel;
            firstblocklevel = 1; // Largest block is 1GB (2^30 bytes)
    stride = grainsize - 3; // Log2(page size / 8 bytes)

    // Limits on IPA controls based on implemented PA size. Level 0 is only

```

```

// supported by small grain translations
if largegrain then // 64KB pages
    // Level 1 only supported if implemented PA size is greater than 2^42 bytes
    if level == 0 || (level == 1 && PAMax() <= 42) then basefound = FALSE;
elseif midgrain then // 16KB pages
    // Level 1 only supported if implemented PA size is greater than 2^40 bytes
    if level == 0 || (level == 1 && PAMax() <= 40) then basefound = FALSE;
else // Small grain, 4KB pages
    // Level 0 only supported if implemented PA size is greater than 2^42 bytes
    if level < 0 || (level == 0 && PAMax() <= 42) then basefound = FALSE;

// If the inputsize exceeds the PAMax value, the behavior is CONSTRAINED UNPREDICTABLE
inputsizecheck = inputsize;
if inputsize > PAMax() && (!ELUsingAArch32(EL1) || inputsize > 40) then
    case ConstrainUnpredictable(Unpredictable_LARGEIPA) of
        when Constraint_FORCE
            // Restrict the inputsize to the PAMax value
            inputsize = PAMax();
            inputsizecheck = PAMax();
        when Constraint_FORCENOSLCHECK
            // As FORCE, except use the configured inputsize in the size checks below
            inputsize = PAMax();
        when Constraint_FAULT
            // Generate a translation fault
            basefound = FALSE;
        otherwise
            Unreachable();

// Number of entries in the starting level table =
// (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
startsizecheck = inputsizecheck - ((3 - level)*stride + grainsize); // Log2(Num of entries)

// Check for starting level table with fewer than 2 entries or longer than 16 pages.
// Lower bound check is: startsizecheck < Log2(2 entries)
// Upper bound check is: startsizecheck > Log2(pagesize/8*16)
if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

if !basefound || disabled then
    level = 0; // AArch32 reports this as a level 1 fault
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswri
        secondstage, s2fslwalk);
    return result;

case ps of
    when '000' outputsize = 32;
    when '001' outputsize = 36;
    when '010' outputsize = 40;
    when '011' outputsize = 42;
    when '100' outputsize = 44;
    when '101' outputsize = 48;
    when '110' outputsize = (if Have52BitPAExt() && largegrain then 52 else 48);
    otherwise outputsize = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address

if outputsize > PAMax() then outputsize = PAMax();

if outputsize < 48 && !IsZero(baseregister<47:outputsize>) then
    level = 0;
    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, s1_nonsecure, level, acctype, iswri
        secondstage, s2fslwalk);
    return result;

// Bottom bound of the Base address is:
// Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
// (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize); // Log2(Num of entries*8)
if outputsize == 52 then
    z = (if baselowerbound < 6 then 6 else baselowerbound);
    baseaddress = baseregister<5:2>:baseregister<47:z>:Zeros(z);
else

```

```

        baseaddress = ZeroExtend(baseregister<47:baselowerbound>:Zeros(baselowerbound));

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';

addrselecttop = inputsized - 1;

apply_nvnvl_effect = HaveNVExt() && EL2EnabledHaveEL() && HCR_EL2.<NV,NV1> == '11' && S1Translation

repeat
    addrselectbottom = (3-level)*stride + grainsize;

    bits(52) index = EL1 && !secondstage;
repeat
    addrselectbottom = (3-level)*stride + grainsize;

    bits(52) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
    descaddr.paddress.address = baseaddress OR index;
    descaddr.paddress.NS = ns_table;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if secondstage || !HasS2Translation() || (HaveNV2Ext() && acctype == AccType_NV2REGISTER) then
        descaddr2 = descaddr;
    else
        hwupdatewalk = FALSE;
        descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
        // Check for a fault on the stage 2 walk
        if IsFault(descaddr2) then
            result.addrdesc.fault = descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    descaddr2.vaddress = ZeroExtend(vaddress);

    accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fslwalk, level);
    desc = _Mem[descaddr2, 8, accdesc];

    if reversedescriptors then desc = accdesc = CreateAccessDescriptorPTW(acctype, secondstage,
    desc = _Mem[descaddr2, 8, accdesc];

    if reversedescriptors then desc = BigEndianReverse(desc);

    if desc<0> == '0' || (desc<1:0> == '01' && (level == 3 ||
        (HaveBlockBBM() && IsBlockDescriptorNTBitValid() &&
        // Fault (00), Reserved (10), Block (01) at level 3, or Block(01) with nT bit set.
(desc);

    if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
        // Fault (00), Reserved (10), or Block (01) at level 3
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype,
            iswrite, secondstage, s2fslwalk);

        return result;

    // Valid Block, Page, or Table entry
    if desc<1:0> == '01' || level == 3 then // Block (01) or Page (11)
        blocktranslate = TRUE;
    else // Table (11)
        if (outputsize < 52 && largegrain && !IsZero(desc<15:12>)) || (outputsize < 48 && !IsZero(d
            result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, s1_nonsecure, level, acctype,
                iswrite, secondstage, s2fslwalk);

        return result;

    if outputsize == 52 then
        baseaddress = desc<15:12>:desc<47:grainsize>:Zeros(grainsize);
    else
        baseaddress = ZeroExtend(desc<47:grainsize>:Zeros(grainsize));
    if !secondstage then

```

```

        // Unpack the upper and lower table attributes
        ns_table = ns_table OR desc<63>;
    if !secondstage && !hierattrsddisabled then
        ap_table<1> = ap_table<1> OR desc<62>;          // read-only

        if apply_nvnvl_effect then
            pxn_table = pxn_table OR desc<60>;
        else
            xn_table = xn_table OR desc<60>;
        // pxn_table and ap_table[0] apply in EL1&0 or EL2&0 translation regimes
        if !singlepriv then
            if !apply_nvnvl_effect then
                pxn_table = pxn_table OR desc<59>;
                ap_table<0> = ap_table<0> OR desc<61>;    // privileged

            level = level + 1;
            addrselecttop = addrselectbottom - 1;
            blocktranslate = FALSE;
until blocktranslate;

// Check block size is supported at this level
if level < firstblocklevel then
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype,
                                                    iswrite, secondstage, s2fslwalk);

    return result;

// Check for misprogramming of the contiguous bit
if largegrain then
    contiguousbitcheck = level == 2 && inputsizesize < 34;
elseif midgrain then
    contiguousbitcheck = level == 2 && inputsizesize < 30;
else
    contiguousbitcheck = level == 1 && inputsizesize < 34;

if contiguousbitcheck && desc<52> == '1' then
    if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype,
                                                    iswrite, secondstage, s2fslwalk);

        return result;

// Check the output address is inside the supported range
if (outputsizesize < 52 && largegrain && !IsZero(desc<15:12>)) || (outputsizesize < 48 && !IsZero(desc<47:0>)) then
    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, s1_nonsecure, level, acctype,
                                                    iswrite, secondstage, s2fslwalk);

    return result;

// Unpack the descriptor into address and upper and lower block attributes
if outputsizesize == 52 then
    outputaddress = desc<15:12>:desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
else
    outputaddress = ZeroExtend(desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>);
// Check Access Flag
if desc<10> == '0' then
    if !update_AF then
        result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress, s1_nonsecure, level, acctype,
                                                    iswrite, secondstage, s2fslwalk);

        return result;
    else
        result.descupdate.AF = TRUE;

if update_AP && desc<51> == '1' then
    // If hw update of access permission field is configured consider AP[2] as '0' / S2AP[2] as '1'
    if !secondstage && desc<7> == '1' then
        desc<7> = '0';
        result.descupdate.AP = TRUE;
    elseif secondstage && desc<7> == '0' then
        desc<7> = '1';
        result.descupdate.AP = TRUE;

// Required descriptor if AF or AP[2]/S2AP[2] needs update

```

```

result.descupdate.descaddr = descaddr;

if apply_nvnv1_effect then
    pxn = desc<54>; // Bit[54] of the block/page descriptor h
    xn = '0'; // XN is '0'
    ap = desc<7>:'01'; // Bit[6] of the block/page descriptor is
else
    xn = desc<54>; // Bit[54] of the block/page descriptor h
    pxn = desc<53>; // Bit[53] of the block/page descriptor h
    ap = desc<7:6>:'1'; // Bits[7:6] of the block/page descriptor
contiguousbit = desc<52>;
nG = desc<11>;
sh = desc<9:8>;
memattr = desc<5:2>; // AttrIndx and NS bit in stage 1

result.domain = bits(4) UNKNOWN; // Domains not used
result.level = level;
result.blocksize = 2^((3-level)*stride + grainsize);

// Stage 1 translation regimes also inherit attributes from the tables
if !secondstage then
    result.perms.xn = xn OR xn_table;
    result.perms.ap<2> = ap<2> OR ap_table<1>; // Force read-only
    // PXN, nG and AP[1] apply in EL1&0 or EL2&0 stage 1 translation regimes
    if !singlepriv then
        result.perms.ap<1> = ap<1> AND NOT(ap_table<0>); // Force privileged only
        result.perms.pxn = pxn OR pxn_table;
        // Pages from Non-secure tables are marked non-global in Secure EL1&0
        if IsSecure() then
            result.nG = nG OR ns_table;
        else
            result.nG = nG;
    else
        result.perms.ap<1> = '1';
        result.perms.pxn = '0';
        result.nG = '0';
    result.GP = desc<50>; // Stage 1 block or pages might be guarded
    result.perms.ap<0> = '1';
    result.addrdesc.memattr = AArch64.S1AttrDecode(sh, memattr<2:0>, acctype);
    result.addrdesc.paddress.NS = memattr<3> OR ns_table;
else
    result.perms.ap<2:1> = ap<2:1>;
    result.perms.ap<0> = '1';
    result.perms.xn = xn;
    if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
    result.perms.pxn = '0';
    result.nG = '0';
    if s2fslwalk then
        result.addrdesc.memattr = S2AttrDecode(sh, memattr, AccType\_PTW);
    else
        result.addrdesc.memattr = S2AttrDecode(sh, memattr, acctype);
    result.addrdesc.paddress.NS = nsaccess;

result.addrdesc.paddress.address = outputaddress;
result.addrdesc.fault = AArch64.NoFault();
result.contiguous = contiguousbit == '1';
if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;

return result;

```



## Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// =====

ClearStickyErrors()
    EDSCR.TXU = '0';           // Clear TX underrun flag
    EDSCR.RXO = '0';           // Clear RX overrun flag

    if Halted() then           // in Debug state
        EDSCR.ITO = '0';       // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable\_CLEARERRITEZERO) then
        return;
    EDSCR.ERR = '0';           // Clear cumulative error flag

    return;
```

## Library pseudocode for shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =====
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

## Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =====

bits(2) DebugTargetFrom(boolean secure)
    if HaveEL(EL2) && !secure then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```



## Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// =====
// Returns the state of the OS Double Lock.
// FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
// TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if !if HaveDoubleLock() then
        return FALSE;
    elseif ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

## Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// =====
// Returns TRUE if the External Debugger access is allowed.
// Returns the status of EDPRSR.EDAD.

boolean AllowExternalDebugAccess(boolean access_is_secure)
AllowExternalDebugAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if HaveSecureExtDebugViewExternalInvasiveDebugEnabled() || () then
if ExternalDebugEnabledExternalSecureInvasiveDebugEnabled() then
        if HaveSecureExtDebugView() then
            allow_secure = access_is_secure;
        else
            allow_secure = ExternalSecureDebugEnabled();
    if allow_secure then
() then
        return TRUE;
    elseif HaveEL(EL3) then
        return (if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD) == '0';
    else
        return !IsSecure();
    else
        return FALSE;
```

## Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// =====
// Returns TRUE if the External Debugger access is allowed.
// Returns the status of EDPRSR.EPMAD.

boolean AllowExternalPMUAccess(boolean access_is_secure)
AllowExternalPMUAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if HaveSecureExtDebugView() || ExternalNoninvasiveDebugEnabled() then
        if HaveSecureExtDebugView() then
            allow_secure = access_is_secure;
        else
            allow_secure = if ExternalSecureNoninvasiveDebugEnabled();
    if allow_secure then
() then
        return TRUE;
    elseif HaveEL(EL3) then
        return (if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD) == '0';
    else
        return !IsSecure();
    else
        return FALSE;
```

## Library pseudocode for shared/debug/authentication/Debug\_authentication

```
signal DBGEN;  
signal NIDEN;  
signal SPIDEN;  
signal SPNIDEN;
```

## Library pseudocode for shared/debug/authentication/ExternalDebugEnabledExternalInvasiveDebugEnabled

```
// ExternalDebugEnabled()  
// =====  
// ExternalInvasiveDebugEnabled()  
// =====  
  
boolean ExternalDebugEnabled()  
ExternalInvasiveDebugEnabled()  
    // The definition of this function is IMPLEMENTATION DEFINED.  
    // In the recommended interface, this function returns the state of the DBGEN signal.  
// In the recommended interface, ExternalInvasiveDebugEnabled returns the state of the DBGEN  
// signal.  
    return DBGEN == HIGH;
```

## Library pseudocode for shared/debug/authentication/ExternalHypInvasiveDebugEnabled

```
// ExternalHypInvasiveDebugEnabled()  
// =====  
  
boolean ExternalHypInvasiveDebugEnabled()  
    // In the recommended interface, ExternalHypInvasiveDebugEnabled returns the state of the  
    // (DBGEN AND HIDDEN) signal.  
    return ExternalDebugEnabled() && HIDDEN == HIGH;
```

## Library pseudocode for shared/debug/authentication/ExternalHypNoninvasiveDebugEnabled

```
// ExternalHypNoninvasiveDebugEnabled()  
// =====  
  
boolean ExternalHypNoninvasiveDebugEnabled()  
    // The definition of this function is IMPLEMENTATION DEFINED.  
    // In the recommended interface, ExternalHypNoninvasiveDebugEnabled returns the state of the  
    // (DBGEN OR NIDEN) AND (HIDDEN OR HNIDEN) signal.  
    return ExternalNoninvasiveDebugEnabled() && (HIDDEN == HIGH || HNIDEN == HIGH);
```

## Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()  
// =====  
  
boolean ExternalNoninvasiveDebugEnabled()  
    // Return TRUE if Trace and PC Sample-based Profiling are allowed  
    return (ExternalNoninvasiveDebugEnabled() &&  
            (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled() ||  
             (ELUsingAArch32(EL1) && PSTATE.EL == EL0 && SDER.SUNIDEN == '1')));
```

## Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()  
// =====  
  
boolean ExternalNoninvasiveDebugEnabled()  
    // This function returns TRUE if the v8.4 Debug relaxations are implemented, otherwise this  
    // function is IMPLEMENTATION DEFINED.  
// The definition of this function is IMPLEMENTATION DEFINED.  
    // In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN  
    // OR NIDEN) signal.  
    return !return HaveNoninvasiveDebugAuthExternalInvasiveDebugEnabled() || ExternalDebugEnabled();
```

## Library pseudocode for shared/debug/

### authentication/ExternalSecureDebugEnabledExternalSecureInvasiveDebugEnabled

```
// ExternalSecureDebugEnabled()
// =====
// ExternalSecureInvasiveDebugEnabled()
// =====

boolean ExternalSecureDebugEnabled()
ExternalSecureInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, ExternalSecureInvasiveDebugEnabled returns the state of the
    // (DBGEN AND SPIDEN) signal.
    // CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
    // CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.
    return ExternalDebugEnabledExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

### Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =====

boolean ExternalSecureNoninvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, ExternalSecureNoninvasiveDebugEnabled returns the state of the
    // (DBGEN OR NIDEN) AND (SPIDEN OR SPNIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    // If the v8.4 Debug relaxations are implemented, this function returns the value of
    // ExternalSecureDebugEnabled(). Otherwise the definition of this function is
    // IMPLEMENTATION DEFINED.
    // In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
    // (SPIDEN OR SPNIDEN) signal.
    if return HaveNoninvasiveDebugAuth() then
        return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
    else
        return ExternalSecureDebugEnabled(); (-) && (SPIDEN == HIGH || SPNIDEN == HIGH);
```

### Library pseudocode for shared/debug/cti/CTI\_SetEventLevel

```
// Set a Cross Trigger multi-cycle input event trigger to the specified level.
CTI_SetEventLevel(CrossTriggerIn id, signal level);
```

### Library pseudocode for shared/debug/cti/CTI\_SignalEvent

```
// Signal a discrete event on a Cross Trigger input event trigger.
CTI_SignalEvent(CrossTriggerIn id);
```

### Library pseudocode for shared/debug/cti/CrossTrigger

```
enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,           CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,    CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,    CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn {CrossTriggerIn_CrossHalt,        CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,           CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,     CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,     CrossTriggerIn_TraceExtOut3};
```

## Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =====

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    if ELUsingAArch32\(EL1\) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                   (commtx && DBGDCCINT.TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                   (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID\_COMMIRQ, if commirq then HIGH else LOW);

    return;
```

## Library pseudocode for shared/debug/dccanditr/DBGDTRRX\_EL0

```
// DBGDTRRX_EL0[] (external write)
// =====
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

    if EDPRSR<6:5,0> != '001' then                                // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                               // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;           // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1'; EDSCR.ERR = '1';                       // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                         // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xD5330501<31:0>);                       // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401<31:0>);                       // A64 "STR W1,[X0],#4"
            X[1] = bits(64) UNKNOWN;
        else
            ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "STR R1,[R0],#4"
            R[1] = bits(32) UNKNOWN;
        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(32) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
            assert EDSCR.RXfull == '0';

            EDSCR.ITE = '1';                                       // See comments in EDITR[] (external write)
        return;

// DBGDTRRX_EL0[] (external read)
// =====

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
return DTRRX;
```

## Library pseudocode for shared/debug/dccanditr/DBGDTRTX\_EL0

```
// DBGDTRTX_EL0[] (external read)
// =====
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                                // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;                        // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then                    // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1'; EDSCR.ERR = '1';                        // Underrun condition: block side-effects
        return value;                                            // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                        // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);                        // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "LDR R1,[R0],#4"
            // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
            if EDSCR.ERR == '1' then
                EDSCR.TXfull = bit UNKNOWN;
                DBGDTRTX_EL0 = bits(32) UNKNOWN;
            else
                if !UsingAArch32() then
                    ExecuteA64(0xD5130501<31:0>);                // A64 "MSR DBGDTRTX_EL0,X1"
                else
                    ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MSR DBGDTRTXint,R1"
                    // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
                    assert EDSCR.TXfull == '1';

                if !UsingAArch32() then
                    X[1] = bits(64) UNKNOWN;
                else
                    R[1] = bits(32) UNKNOWN;

                EDSCR.ITE = '1';                                    // See comments in EDITR[] (external write)

            return value;

// DBGDTRTX_EL0[] (external write)
// =====

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write
DTRTX = value;
return;
```

## Library pseudocode for shared/debug/dccanditr/DBGDTR\_EL0

```
// DBGDTR_EL0[] (write)
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
// For MSR DBGDTRTX_EL0,<Rt> N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
// For MSR DBGDTR_EL0,<Xt> N=64, value=X[t]<63:0>
assert N IN {32,64};
if EDSCR.TXfull == '1' then
    value = bits(N) UNKNOWN;
// On a 64-bit write, implement a half-duplex channel
if N == 64 then DTRRX = value<63:32>;
DTRTX = value<31:0>; // 32-bit or 64-bit write
EDSCR.TXfull = '1';
return;

// DBGDTR_EL0[] (read)
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
// For MRS <Rt>,DBGDTRTX_EL0 N=32, X[t]=Zeros(32):result
// For MRS <Xt>,DBGDTR_EL0 N=64, X[t]=result
assert N IN {32,64};
bits(N) result;
if EDSCR.RXfull == '0' then
    result = bits(N) UNKNOWN;
else
    // On a 64-bit read, implement a half-duplex channel
    // NOTE: the word order is reversed on reads with regards to writes
    if N == 64 then result<63:32> = DTRTX;
    result<31:0> = DTRRX;
EDSCR.RXfull = '0';
return result;
```

## Library pseudocode for shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

## Library pseudocode for shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// =====
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
  if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    return;

  if EDSCR.ERR == '1' then return; // Error flag set: ignore write

  // The Software lock is OPTIONAL.
  if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write

  if !Halted() then return; // Non-debug state: ignore write

  if EDSCR.ITE == '0' || EDSCR.MA == '1' then
    EDSCR.ITO = '1'; EDSCR.ERR = '1'; // Overrun condition: block write
    return;

  // ITE indicates whether the processor is ready to accept another instruction; the processor
  // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
  // is no indication that the pipeline is empty (all instructions have completed). In this
  // pseudocode, the assumption is that only one instruction can be executed at a time,
  // meaning ITE acts like "InstrCompl".
  EDSCR.ITE = '0';

  if !UsingAArch32() then
    ExecuteA64(value);
  else
    ExecuteT32(value<15:0> /*hw1*/, value<31:16> /*hw2*/);

  EDSCR.ITE = '1';

return;
```





```

// DCPSInstruction()
// =====
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

case target_el of
    when EL1
        if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
        elsif EL2Enabled() && HCR_EL2.TGE == '1' then UndefinedFault();
        else handle_el = EL1;

    when EL2
        if !HaveEL(EL2) then UndefinedFault();
        elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
        elsif !IsSecureEL2Enabled() && IsSecure() then UndefinedFault();
        else handle_el = EL2;

    when EL3
        if EDSCR.SDD == '1' || !HaveEL(EL3) then UndefinedFault();
        handle_el = EL3;
    otherwise
        Unreachable();

from_secure = IsSecure();
if ELUsingAArch32(handle_el) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    assert UsingAArch32(); // Cannot move from AArch64 to AArch32
    case handle_el of
        when EL1AArch32.WriteMode(M32_Svc);
            if HavePANExt() && SCTL.R.SPAN == '0' then
                PSTATE.PAN = '1';
        when EL2_AArch32.WriteMode(M32_Hyp);
        when EL3AArch32.WriteMode(M32_Monitor);
            if HavePANExt() then
                if !from_secure then
                    PSTATE.PAN = '0';
                elsif SCTL.R.SPAN == '0' then
                    PSTATE.PAN = '1';
    if handle_el == EL2 then
        ELR_hyp = bits(32) UNKNOWN; HSR = bits(32) UNKNOWN;
    else
        LR = bits(32) UNKNOWN;
        SPSR[] = bits(32) UNKNOWN;
        PSTATE.E = SCTL.R.EE;
        DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;

else // Targeting AArch64
    if UsingAArch32() then
        AArch64.MaybeZeroRegisterUppers();
        MaybeZeroSVEUppers(target_el);
        PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;
        if HavePANExt() && ((handle_el == EL1 && SCTL.R_EL1.SPAN == '0') ||
            (handle_el == EL2 && HCR_EL2.E2H == '1' &&
            HCR_EL2.TGE == '1' && SCTL.R_EL2.SPAN == '0')) then
            PSTATE.PAN = '1';
        ELR[] = bits(64) UNKNOWN; SPSR[] = bits(32) UNKNOWN; ESR[] = bits(32) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(32) UNKNOWN;

        if HaveUAOExt() then PSTATE.UAO = '0';

    UpdateEDSCRFields(); // Update EDSCR PE state flags
    sync_errors =
    // SCTL.R[].IESB might be ignored in Debug state.
    if HaveIESBHaveRASExt() && SCTL.R[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTL.R[].IESB might be ignored in Debug state.
    if ![].IESB == '1' && ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then

```

```

    sync_errors = FALSE;
    if sync_errors then then
        SynchronizeErrors();
    return;

```

## Library pseudocode for shared/debug/halting/DRPSInstruction

```

// DRPSInstruction()
// =====
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    sync_errors = FALSE // SCTLR[].IESB might be ignored in Debug state.
    if HaveIESBHaveRASExt() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLR[].IESB might be ignored in Debug state.
    if ![].IESB == '1' && ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then then
        SynchronizeErrors();

    SetPSTATEFromPSR(SPSR[]);

    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        PSTATE.<N,Z,C,V,Q,GE,SS,D,A,I,F> = bits(13) UNKNOWN;
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;
    else
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(32) UNKNOWN;

    UpdateEDSCRFields(); // Update EDSCR PE state flags

    return;

```

## Library pseudocode for shared/debug/halting/DebugHalt

```

constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';

```

## Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```

DisableITRAndResumeInstructionPrefetch();

```

## Library pseudocode for shared/debug/halting/ExecuteA64

```

// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);

```

## Library pseudocode for shared/debug/halting/ExecuteT32

```
// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);
```

## Library pseudocode for shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// =====

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001'; // Signal restarting
    EDESR<2:0> = '000'; // Clear any pending Halting debug events

    bits(64) new_pc;
    bits(32) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR);
        spsr = DSPSR;
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr); // Can update privileged bits, even at EL0

    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
        BranchTo(new_pc<31:0>, BranchType_DBGEXITBranchType_UNKNOWN); // AArch32 branch
    else
        // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
        if spsr<4> == '1' && ConstrainUnpredictableBool(Unpredictable_RESTARTZEROUPPERPC) then
            new_pc<63:32> = Zeros();
            BranchTo(new_pc, BranchType_DBGEXIT); // A type of branch that is never predicted

    (EDSCR.STATUS, EDPRSR.SDR) = ('000010', '1'); // Atomically signal restarted
    UpdateEDSCRFields(); // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
```

## Library pseudocode for shared/debug/halting/Halt

```
// Halt()
// =====

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt); // Trigger other cores to halt
    if UsingAArch32() then
        DLR = ThisInstrAddr();
        DSPSR = GetPSRFromPSTATE();
        DSPSR.SS = PSTATE.SS; // Always save PSTATE.SS
    else
        DLR_EL0 = ThisInstrAddr();
        DSPSR_EL0 = GetPSRFromPSTATE();
        DSPSR_EL0.SS = PSTATE.SS; // Always save PSTATE.SS

    EDSCR.ITE = '1'; EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0'; // If entered in Secure state, allow debug
    elseif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureDebugEnabledExternalSecureInvasiveDebugEnabled() then '0' else '1'
    else
        assert EDSCR.SDD == '1'; // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';
    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if
    // UNKNOWN. PSTATE.{N,Z,C,V,Q,GE} are also not observable, but since these are not changed on
    // exception entry, this function also leaves them unchanged. PSTATE.{E,M,nRW,EL,SP} are
    // unchanged. PSTATE.IL is set to 0.
    if UsingAArch32() then
        PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
    else
        PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
        PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason; // Signal entered Debug state
    UpdateEDSCRFields(); // Update EDSCR PE state flags.

    return;
```

## Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// =====
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

## Library pseudocode for shared/debug/halting/Halted

```
// Halted()
// =====

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'}); // Halted
```

## Library pseudocode for shared/debug/halting/HaltingAllowed

```
// HaltingAllowed()
// =====
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elseif IsSecure() then
        return ExternalSecureDebugEnabledExternalSecureInvasiveDebugEnabled();
    else
        return ExternalDebugEnabledExternalInvasiveDebugEnabled();
```

## Library pseudocode for shared/debug/halting/Restarting

```
// Restarting()
// =====

boolean Restarting()
    return EDSCR.STATUS == '000001'; // Restarting
```

## Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

```
StopInstructionPrefetchAndEnableITR();
```

## Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// =====
// Update EDSCR PE state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;
        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        EDSCR.NS = if IsSecure() then '0' else '1';

        bits(4) RW;
        RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW<0> = RW<1>;
        else
            RW<0> = if UsingAArch32() then '0' else '1';
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0' && !IsSecureEL2Enabled()) then
            RW<2> = RW<1>;
        else
            RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
        if !HaveEL(EL3) then
            RW<3> = RW<2>;
        else
            RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elseif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elseif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;
    return;
```

## Library pseudocode for shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that IsSecure() and PSTATE.EL are correct
    // for the exception target.
    base = if IsSecure() then 0 else 4;
    if HaltingAllowed() then
        if HaveExtendedECDebugEvents() then
            exception_exit = !exception_entry;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + 8>; EDECCR<UInt(PSTATE.EL) + base>;
            case ctrl of
                when '00' halt = FALSE;
                when '01' halt = TRUE;
                when '10' halt = (exception_exit == TRUE);
                when '11' halt = (exception_entry == TRUE);
            else
                halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');
        if halt then Halt(DebugHalt\_ExceptionCatch);
```

## Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// =====
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep\_DidNotStep() then
            Halt(DebugHalt\_Step\_NoSyndrome);
        elsif HaltingStep\_SteppedEX() then
            Halt(DebugHalt\_Step\_Exclusive);
        else
            Halt(DebugHalt\_Step\_Normal);
```

## Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if EDECR.OSUCE == '1' && !Halted() then EDESR.OSUC = '1';
```

## Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// =====
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt\_OSUnlockCatch);
```

## Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// =====
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt\_ResetCatch);
```

## Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =====
// Called after reset

CheckResetCatch()
    if EDECR.RCE == '1' then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt\_ResetCatch);
```

## Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt\_SoftwareAccess);
```

## Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// =====

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt(DebugHalt\_EDBGRQ);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

## Library pseudocode for shared/debug/haltingevents/HaltingStep\_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();
```

## Library pseudocode for shared/debug/haltingevents/HaltingStep\_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();
```



## Library pseudocode for shared/debug/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// =====

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
                boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    //
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exception.
    //
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted(); // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();

    if active && reset then // Coming out of reset with EDECR.SS set
        EDESR.SS = '1';
    elseif active && HaltingAllowed() then
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureDebugEnabledExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

## Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =====
// Determine whether EDSCR disables interrupts routed to 'target'

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    case target of
        when EL3
            int_dis = EDSCR.INTdis == '11' && ExternalSecureDebugEnabledExternalSecureInvasiveDebugEnabled();
        when EL2
            int_dis = EDSCR.INTdis == '1x' && ExternalDebugEnabledExternalInvasiveDebugEnabled();
        when EL1
            if IsSecure() then
                int_dis = EDSCR.INTdis == '1x' && ExternalSecureDebugEnabledExternalSecureInvasiveDebugEnabled();
            else
                int_dis = EDSCR.INTdis != '00' && ExternalDebugEnabledExternalInvasiveDebugEnabled();
    return int_dis;
```

## Library pseudocode for shared/debug/interrupts/InterruptID

```
enumeration InterruptID {InterruptID_PMUIRQ, InterruptID_COMMIRQ, InterruptID_CTIIRQ,
                        InterruptID_COMMRX, InterruptID_COMMTX};
```

## Library pseudocode for shared/debug/interrupts/SetInterruptRequestLevel

```
// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);
```

## Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// =====

CreatePCSample()
// In a simple sequential execution of the program, CreatePCSample is executed each time the PE
// executes an instruction that can be sampled. An implementation is not constrained such that
// reads of EDPCSRlo return the current values of PC, etc.

pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
pc_sample.pc = ThisInstrAddr();
pc_sample.el = PSTATE.EL;
pc_sample.rw = if UsingAArch32() then '0' else '1';
pc_sample.ns = if IsSecure() then '0' else '1';
pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1;
pc_sample.has_el2 = EL2Enabled();
if EL2Enabled() then
    if ELUsingAArch32(EL2) then
        pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
    elseif !Have16bitVMID() || VTCR_EL2.VS == '0' then
        pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    else
        pc_sample.vmid = VTTBR_EL2.VMID;
    if HaveVirtHostExt() && !ELUsingAArch32(EL2) then
        pc_sample.contextidr_el2 = CONTEXTIDR_EL2;
    else
        pc_sample.contextidr_el2 = bits(32) UNKNOWN;
    pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
return;
```

## Library pseudocode for shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// =====

bits(32) EDPCSRlo[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0'; // Software locked: no side-effects

if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        if HaveVirtHostExt() && EDSCR.SC2 == '1' then
            EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            EDPCSRhi.EL = pc_sample.el;
            EDPCSRhi.NS = pc_sample.ns;
        else
            EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDVIDSR = (if HaveEL(EL2) && pc_sample.ns == '1' then pc_sample.contextidr_el2
                    else bits(32) UNKNOWN);
            else
                if HaveEL(EL2) && pc_sample.ns == '1' && pc_sample.el IN {EL1,EL0} then
                    EDVIDSR.VMID = pc_sample.vmid;
                else
                    EDVIDSR.VMID = Zeros();
                    EDVIDSR.NS = pc_sample.ns;
                    EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
                    EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
                    // The conditions for setting HV are not specified if PCSRhi is zero.
                    // An example implementation may be "pc_sample.rw".
                    EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
        else
            sample = Ones(32);
            if update then
                EDPCSRhi = bits(32) UNKNOWN;
                EDCIDSR = bits(32) UNKNOWN;
                EDVIDSR = bits(32) UNKNOWN;

    return sample;
```

## Library pseudocode for shared/debug/samplebasedprofiling/PCSample

```
type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    bit ns,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)

PCSample pc_sample;
```

## Library pseudocode for shared/debug/samplebasedprofiling/PMPCSR

```
// PMPCSR[] (read)
// =====

bits(32) PMPCSR[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || PMLSR.SLK == '0'; // Software locked: no side-effects

if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
        PMPCSR.EL = pc_sample.el;
        PMPCSR.NS = pc_sample.ns;

        PMCID1SR = pc_sample.contextidr;
        PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;

        PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h
            then pc_sample.vmid else bits(16) UNKNOWN);
    else
        sample = Ones(32);
        if update then
            PMPCSR<55:32> = bits(24) UNKNOWN;
            PMPCSR.EL = bits(2) UNKNOWN;
            PMPCSR.NS = bit UNKNOWN;

            PMCID1SR = bits(32) UNKNOWN;
            PMCID2SR = bits(32) UNKNOWN;

            PMVIDSR.VMID = bits(16) UNKNOWN;

return sample;
```

## Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// =====
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

// Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
// AArch32 state. However, because Software Step is only active when the debug target Exception
// level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
if !ELUsingAArch32\(DebugTarget\(\)\) && AArch64.GenerateDebugExceptions\(\) then
    if MDSCR_EL1.SS == '1' && PSTATE.SS == '0' then
        AArch64.SoftwareStepException\(\);
```

## Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// =====
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(32) spsr)
    assert Halted() || Restarting() || PSTATE.EL != EL0;

    SS_bit = '0';

    if MDSCR_EL1.SS == '1' then
        if Restarting() then
            enabled_at_source = FALSE;
        elsif UsingAArch32() then
            enabled_at_source = AArch32.GenerateDebugExceptions();
        else
            enabled_at_source = AArch64.GenerateDebugExceptions();

        if IllegalExceptionReturn(spsr) then
            dest = PSTATE.EL;
        else
            (valid, dest) = ELFromSPSR(spsr); assert valid;

        secure = IsSecureBelowEL3() || dest == EL3;

        if ELUsingAArch32(dest) then
            enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
        else
            mask = spsr<9>;
            enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);

        ELd = DebugTargetFrom(secure);
        if !ELUsingAArch32(ELd) && !enabled_at_source && enabled_at_dest then
            SS_bit = spsr<21>;
    return SS_bit;
```

## Library pseudocode for shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// =====
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

## Library pseudocode for shared/debug/softwarestep/SoftwareStep\_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean SoftwareStep_DidNotStep();
```

## Library pseudocode for shared/debug/softwarestep/SoftwareStep\_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean SoftwareStep_SteppedEX();
```

## Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// =====
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    if UsingAArch32\(\) then
        cond = AArch32.CurrentCond\(\);
        if PSTATE.T == '0' then // A32
            syndrome<4> = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable\_ESRCONDPASS) then
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            // * CV set to 0 and COND is set to an UNKNOWN value
            // * CV set to 1 and COND is set to the condition code for the condition that
            //   applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';

    return syndrome;
```

## Library pseudocode for shared/exceptions/exceptions/Exception

```

enumeration Exception {Exception_Uncategorized,      // Uncategorized or unknown reason
    Exception_WFxTrap,      // Trapped WFI or WFE instruction
    Exception_CP15RTTTrap,  // Trapped AArch32 MCR or MRC access to CP15
    Exception_CP15RRTTrap,  // Trapped AArch32 MCRR or MRRC access to CP15
    Exception_CP14RTTTrap,  // Trapped AArch32 MCR or MRC access to CP14
    Exception_CP14DTTTrap,  // Trapped AArch32 LDC or STC access to CP14
    Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
    Exception_FPIDTrap,     // Trapped access to SIMD or FP ID register
    // Trapped BXJ instruction not supported in ARMv8
    Exception_PACTrap,      // Trapped invalid PAC use
    Exception_CP14RRTTrap,  // Trapped MRRC access to CP14 from AArch32
    Exception_IllegalState, // Illegal Execution state
    Exception_SupervisorCall, // Supervisor Call
    Exception_HypervisorCall, // Hypervisor Call
    Exception_MonitorCall,   // Monitor Call or Trapped SMC instruction
    Exception_SystemRegisterTrap, // Trapped MRS or MSR system register access
    Exception_ERetTrap,      // Trapped invalid ERET use
    Exception_InstructionAbort, // Instruction Abort or Prefetch Abort
    Exception_PCAalignment,   // PC alignment fault
    Exception_DataAbort,     // Data Abort
    Exception_NV2DataAbort,   // Data abort at EL1 reported as being from EL2
    Exception_SPAalignment,   // SP alignment fault
    Exception_FPTrappedException, // IEEE trapped FP exception
    Exception_SError,        // SError interrupt
    Exception_Breakpoint,    // (Hardware) Breakpoint
    Exception_SoftwareStep,   // Software Step
    Exception_Watchpoint,    // Watchpoint
    Exception_SoftwareBreakpoint, // Software Breakpoint Instruction
    Exception_VectorCatch,    // AArch32 Vector Catch
    Exception_IRQ,           // IRQ interrupt
    Exception_SVEAccessTrap,  // HCPTR trapped access to SVE
    Exception_BranchTarget,   // Branch Target Identification
    Exception_FIQ};          // FIQ interrupt

```

## Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```

type ExceptionRecord is (Exception type,      // Exception class
    bits(25) syndrome,      // Syndrome record
    bits(64) vaddress,      // Virtual fault address
    boolean ipavalid,       // Physical fault address for second stage faults
    bits(1) NS,             // Physical fault address for second stage faults
    bits(52) ipaddress)     // Physical fault address for second stage faults

```

## Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```

// ExceptionSyndrome()
// =====
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception type)

    ExceptionRecord r;

    r.type = type;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.NS = '0';
    r.ipaddress = Zeros();

    return r;

```

## Library pseudocode for shared/exceptions/traps/ReservedValue

```
// ReservedValue()
// =====

ReservedValue()
  if UsingAArch32\(\) && !AArch32.GeneralExceptionsToAArch64\(\) then
    AArch32.TakeUndefInstrException\(\);
  else
    AArch64.UndefinedFault\(\);
```

## Library pseudocode for shared/exceptions/traps/SystemAccessType

```
enumeration SystemAccessType { SystemAccessType_RT, SystemAccessType_RRT, SystemAccessType_DT };
```

## Library pseudocode for shared/exceptions/traps/UnallocatedEncoding

```
// UnallocatedEncoding()
// =====

UnallocatedEncoding()
  if UsingAArch32\(\) && AArch32.ExecutingCP10or11Instr\(\) then
    FPExc.DEX = '0';
  if UsingAArch32\(\) && !AArch32.GeneralExceptionsToAArch64\(\) then
    AArch32.TakeUndefInstrException\(\);
  else
    AArch64.UndefinedFault\(\);
```

## Library pseudocode for shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =====
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault type, integer level)

  bits(6) result;
  case type of
    when Fault AddressSize      result = '0000':level<1:0>; assert level IN {0,1,2,3};
    when Fault AccessFlag      result = '0010':level<1:0>; assert level IN {1,2,3};
    when Fault Permission     result = '0011':level<1:0>; assert level IN {1,2,3};
    when Fault Translation    result = '0001':level<1:0>; assert level IN {0,1,2,3};
    when Fault SyncExternal   result = '010000';
    when Fault SyncExternalOnWalk result = '0101':level<1:0>; assert level IN {0,1,2,3};
    when Fault SyncParity     result = '011000';
    when Fault SyncParityOnWalk result = '0111':level<1:0>; assert level IN {0,1,2,3};
    when Fault AsyncParity    result = '011001';
    when Fault AsyncExternal  result = '010001';
    when Fault Alignment     result = '100001';
    when Fault Debug         result = '100010';
    when Fault TLBConflict    result = '110000';
    when Fault HWUpdateAccessFlag result = '110001';
    when Fault Lockdown     result = '110100'; // IMPLEMENTATION DEFINED
    when Fault Exclusive     result = '110101'; // IMPLEMENTATION DEFINED
    otherwise                 Unreachable\(\);

  return result;
```



## Library pseudocode for shared/functions/aborts/IPAValid

```
// IPAValid()
// =====
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.type != Fault\_None;

    if fault.s2fslwalk then
        return fault.type IN {Fault\_AccessFlag, Fault\_Permission, Fault\_Translation,
                               Fault\_AddressSize};
    elsif fault.secondstage then
        return fault.type IN {Fault\_AccessFlag, Fault\_Translation, Fault\_AddressSize};
    else
        return FALSE;
```

## Library pseudocode for shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault type)
    assert type != Fault\_None;

    return (type IN {Fault\_AsyncExternal, Fault\_AsyncParity});

// IsAsyncAbort()
// =====

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.type);
```

## Library pseudocode for shared/functions/aborts/IsDebugException

```
// IsDebugException()
// =====

boolean IsDebugException(FaultRecord fault)
    assert fault.type != Fault\_None;
    return fault.type == Fault\_Debug;
```

## Library pseudocode for shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =====
// Returns TRUE if the abort currently being processed is an external abort and FALSE otherwise.

boolean IsExternalAbort(Fault type)
    assert type != Fault\_None;

    return (type IN {Fault\_SyncExternal, Fault\_SyncParity, Fault\_SyncExternalOnWalk, Fault\_SyncParityOnWalk,
                     Fault\_AsyncExternal, Fault\_AsyncParity });

// IsExternalAbort()
// =====

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.type);
```

### Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault type)
    assert type != Fault\_None;

    return (type IN {Fault\_SyncExternal, Fault\_SyncParity, Fault\_SyncExternalOnWalk, Fault\_SyncParityOnWalk});

// IsExternalSyncAbort()
// =====

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.type);
```

### Library pseudocode for shared/functions/aborts/IsFault

```
// IsFault()
// =====
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.type != Fault\_None;
```

### Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// =====
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE otherwise.

boolean IsSErrorInterrupt(Fault type)
    assert type != Fault\_None;

    return (type IN {Fault\_AsyncExternal, Fault\_AsyncParity});

// IsSErrorInterrupt()
// =====

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.type);
```

### Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// =====

boolean IsSecondStage(FaultRecord fault)
    assert fault.type != Fault\_None;

    return fault.secondstage;
```

### Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

```
bits(11) LSInstructionSyndrome();
```

## Library pseudocode for shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR\_C(x, shift);
    return result;
```

## Library pseudocode for shared/functions/common/ASR\_C

```
// ASR_C()
// =====

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    shift = if shift > N then N else shift;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

## Library pseudocode for shared/functions/common/Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

## Library pseudocode for shared/functions/common/Align

```
// Align()
// =====

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =====

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

## Library pseudocode for shared/functions/common/BitCount

```
// BitCount()
// =====

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

## Library pseudocode for shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// =====

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

## Library pseudocode for shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// =====

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);
```

## Library pseudocode for shared/functions/common/Elem

```
// Elem[] - non-assignment form
// =====

bits(size) Elem(bits(N) vector, integer e, integer size)
    assert e >= 0 && (e+1)*size <= N;
    return vector<e*size+size-1 : e*size>;

// Elem[] - non-assignment form
// =====

bits(size) Elem(bits(N) vector, integer e)
    return Elem[vector, e, size];

// Elem[] - assignment form
// =====

Elem(bits(N) &vector, integer e, integer size) = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;

// Elem[] - assignment form
// =====

Elem(bits(N) &vector, integer e) = bits(size) value
    Elem[vector, e, size] = value;
    return;
```

## Library pseudocode for shared/functions/common/Extend

```
// Extend()
// =====

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

// Extend()
// =====

bits(N) Extend(bits(M) x, boolean unsigned)
    return Extend(x, N, unsigned);
```

### Library pseudocode for shared/functions/common/HighestSetBit

```
// HighestSetBit()
// =====

integer HighestSetBit(bits(N) x)
  for i = N-1 downto 0
    if x<i> == '1' then return i;
  return -1;
```

### Library pseudocode for shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
  result = if unsigned then UInt(x) else SInt(x);
  return result;
```

### Library pseudocode for shared/functions/common/IsOnes

```
// IsOnes()
// =====

boolean IsOnes(bits(N) x)
  return x == Ones(N);
```

### Library pseudocode for shared/functions/common/IsZero

```
// IsZero()
// =====

boolean IsZero(bits(N) x)
  return x == Zeros(N);
```

### Library pseudocode for shared/functions/common/IsZeroBit

```
// IsZeroBit()
// =====

bit IsZeroBit(bits(N) x)
  return if IsZero(x) then '1' else '0';
```

### Library pseudocode for shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
  assert shift >= 0;
  if shift == 0 then
    result = x;
  else
    (result, -) = LSL\_C(x, shift);
  return result;
```

## Library pseudocode for shared/functions/common/LSL\_C

```
// LSL_C()
// =====

(bits(N), bit) LSL_C(bits(N) x, integer shift)
  assert shift > 0;
  shift = if shift > N then N else shift;
  extended_x = x : Zeros(shift);
  result = extended_x<N-1:0>;
  carry_out = extended_x<N>;
  return (result, carry_out);
```

## Library pseudocode for shared/functions/common/LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
  assert shift >= 0;
  if shift == 0 then
    result = x;
  else
    (result, -) = LSR\_C(x, shift);
  return result;
```

## Library pseudocode for shared/functions/common/LSR\_C

```
// LSR_C()
// =====

(bits(N), bit) LSR_C(bits(N) x, integer shift)
  assert shift > 0;
  shift = if shift > N then N else shift;
  extended_x = ZeroExtend(x, shift+N);
  result = extended_x<shift+N-1:shift>;
  carry_out = extended_x<shift-1>;
  return (result, carry_out);
```

## Library pseudocode for shared/functions/common/LowestSetBit

```
// LowestSetBit()
// =====

integer LowestSetBit(bits(N) x)
  for i = 0 to N-1
    if x<i> == '1' then return i;
  return N;
```

## Library pseudocode for shared/functions/common/Max

```
// Max()
// =====

integer Max(integer a, integer b)
  return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
  return if a >= b then a else b;
```

## Library pseudocode for shared/functions/common/Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```

## Library pseudocode for shared/functions/common/Ones

```
// Ones()
// =====

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// =====

bits(N) Ones()
    return Ones(N);
```

## Library pseudocode for shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR\_C(x, shift);
    return result;
```

## Library pseudocode for shared/functions/common/ROR\_C

```
// ROR_C()
// =====

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

## Library pseudocode for shared/functions/common/Replicate

```
// Replicate()
// =====

bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);
```

### Library pseudocode for shared/functions/common/RoundDown

```
integer RoundDown(real x);
```

### Library pseudocode for shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// =====

integer RoundTowardsZero(real x)
  return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

### Library pseudocode for shared/functions/common/RoundUp

```
integer RoundUp(real x);
```

### Library pseudocode for shared/functions/common/SInt

```
// SInt()
// =====

integer SInt(bits(N) x)
  result = 0;
  for i = 0 to N-1
    if x<i> == '1' then result = result + 2^i;
  if x<N-1> == '1' then result = result - 2^N;
  return result;
```

### Library pseudocode for shared/functions/common/SignExtend

```
// SignExtend()
// =====

bits(N) SignExtend(bits(M) x, integer N)
  assert N >= M;
  return Replicate(x<M-1>, N-M) : x;

// SignExtend()
// =====

bits(N) SignExtend(bits(M) x)
  return SignExtend(x, N);
```

### Library pseudocode for shared/functions/common/UInt

```
// UInt()
// =====

integer UInt(bits(N) x)
  result = 0;
  for i = 0 to N-1
    if x<i> == '1' then result = result + 2^i;
  return result;
```



### Library pseudocode for shared/functions/common/ZeroExtend

```
// ZeroExtend()
// =====

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;

// ZeroExtend()
// =====

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

### Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =====

bits(N) Zeros(integer N)
    return Replicate('0',N);

// Zeros()
// =====

bits(N) Zeros()
    return Zeros(N);
```

### Library pseudocode for shared/functions/crc/BitReverse

```
// BitReverse()
// =====

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<N-i-1> = data<i>;
    return result;
```

### Library pseudocode for shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// =====

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

### Library pseudocode for shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// =====

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
    assert N > 32;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```

### Library pseudocode for shared/functions/crypto/AESInvMixColumns

```
bits(128) AESInvMixColumns(bits (128) op);
```

### Library pseudocode for shared/functions/crypto/AESInvShiftRows

```
bits(128) AESInvShiftRows(bits(128) op);
```

### Library pseudocode for shared/functions/crypto/AESInvSubBytes

```
bits(128) AESInvSubBytes(bits(128) op);
```

### Library pseudocode for shared/functions/crypto/AESMixColumns

```
bits(128) AESMixColumns(bits (128) op);
```

### Library pseudocode for shared/functions/crypto/AESShiftRows

```
bits(128) AESShiftRows(bits(128) op);
```

### Library pseudocode for shared/functions/crypto/AESSubBytes

```
bits(128) AESSubBytes(bits(128) op);
```

### Library pseudocode for shared/functions/crypto/HaveAESExt

```
// HaveAESExt()  
// =====  
// TRUE if AES cryptographic instructions support is implemented,  
// FALSE otherwise.  
  
boolean HaveAESExt()  
    return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveBit128PMULLExt

```
// HaveBit128PMULLExt()  
// =====  
// TRUE if 128 bit form of PMULL instructions support is implemented,  
// FALSE otherwise.  
  
boolean HaveBit128PMULLExt()  
    return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSHA1Ext

```
// HaveSHA1Ext()  
// =====  
// TRUE if SHA1 cryptographic instructions support is implemented,  
// FALSE otherwise.  
  
boolean HaveSHA1Ext()  
    return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSHA256Ext

```
// HaveSHA256Ext()  
// =====  
// TRUE if SHA256 cryptographic instructions support is implemented,  
// FALSE otherwise.  
  
boolean HaveSHA256Ext()  
    return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSHA3Ext

```
// HaveSHA3Ext()
// =====
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    if !HasArchVersion(ARMv8p2) || !(HaveSHA1Ext() && HaveSHA256Ext()) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSHA512Ext

```
// HaveSHA512Ext()
// =====
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    if !HasArchVersion(ARMv8p2) || !(HaveSHA1Ext() && HaveSHA256Ext()) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSM3Ext

```
// HaveSM3Ext()
// =====
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/HaveSM4Ext

```
// HaveSM4Ext()
// =====
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";
```

### Library pseudocode for shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

### Library pseudocode for shared/functions/crypto/SHA256hash

```
// SHA256hash()
// =====

bits(128) SHA256hash(bits (128) X, bits(128) Y, bits(128) W, boolean part1)
    bits(32) chs, maj, t;

    for e = 0 to 3
        chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
        maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
        t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
        X<127:96> = t + X<127:96>;
        Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
        <Y, X> = ROL(Y : X, 32);
    return (if part1 then X else Y);
```

### Library pseudocode for shared/functions/crypto/SHAchoose

```
// SHAchoose()
// =====

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return ((y EOR z) AND x) EOR z;
```

### Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// =====

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

### Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// =====

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

### Library pseudocode for shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =====

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));
```

### Library pseudocode for shared/functions/crypto/SHAparity

```
// SHAparity()
// =====

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

## Library pseudocode for shared/functions/crypto/Sbox

```
// Sbox()
// =====
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = 0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f49

    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

## Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

## Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```

## Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// =====

// Clear the local Exclusives monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

## Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

```
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();
```

## Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

```
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

## Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

```
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

## Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

### Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// =====

boolean AArch32.HaveHPDExt()
    return HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// =====

boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

### Library pseudocode for shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// =====

boolean Have52BitPAExt()
    return HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// =====

boolean Have52BitVAExt()
    return HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// =====

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);
```

### Library pseudocode for shared/functions/extension/HaveBTIExt

```
// HaveBTIExt()
// =====
// Returns TRUE if BTI implemented and FALSE otherwise

boolean HaveBTIExt()
    return HasArchVersion(ARMv8p5);
```

### Library pseudocode for shared/functions/extension/HaveBlockBBM

```
// HaveBlockBBM()
// =====
// Returns TRUE if support for changing block size without requiring break-before-make is implemented.

boolean HaveBlockBBM()
    return HasArchVersion(ARMv8p4);
```

### Library pseudocode for shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// =====

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/HaveDITExt

```
// HaveDITExt()
// =====

boolean HaveDITExt()
    return HasArchVersion(ARMv8p4);
```

### Library pseudocode for shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()
// =====
// Returns TRUE if has Dot Product feature support, and FALSE otherwise.

boolean HaveDOTPExt()
    return HasArchVersion(ARMv8p4) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has D
```

### Library pseudocode for shared/functions/extension/HaveDoubleFaultExt

```
// HaveDoubleFaultExt()
// =====

boolean HaveDoubleFaultExt()
    return (HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESB());
```

### Library pseudocode for shared/functions/extension/HaveDoubleLock

```
// HaveDoubleLock()
// =====
// Returns TRUE if support for the OS Double Lock is implemented

boolean HaveDoubleLock()
    return !HasArchVersion(ARMv8p4) || boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented";
```

### Library pseudocode for shared/functions/extension/HaveE0PDExt

```
// HaveE0PDExt()
// =====
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.

boolean HaveE0PDExt()
    return HasArchVersion(ARMv8p5);
```

## Library pseudocode for shared/functions/extension/HaveExtendedCacheSets

```
// HaveExtendedCacheSets ()
// =====

boolean HaveExtendedCacheSets ()
    return HasArchVersion (ARMv8p3);
```

## Library pseudocode for shared/functions/extension/HaveExtendedECDebugEvents

```
// HaveExtendedECDebugEvents ()
// =====

boolean HaveExtendedECDebugEvents ()
    return HasArchVersion (ARMv8p2);
```

## Library pseudocode for shared/functions/extension/HaveExtendedExecuteNeverExt

```
// HaveExtendedExecuteNeverExt ()
// =====

boolean HaveExtendedExecuteNeverExt ()
    return HasArchVersion (ARMv8p2);
```

## Library pseudocode for shared/functions/extension/HaveFCADDExt

```
// HaveFCADDExt ()
// =====

boolean HaveFCADDExt ()
    return HasArchVersion (ARMv8p3);
```

## Library pseudocode for shared/functions/extension/HaveJCVTZSExt

```
// HaveJCVTZSExt ()
// =====

boolean HaveJCVTZSExt ()
    return HasArchVersion (ARMv8p3);
```

## Library pseudocode for shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```
// HaveFP16MulNoRoundingToFP32Ext ()
// =====
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate to FP32 instructions,
// and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext ()

    if !HaveFP16Ext() then return FALSE;
    if HasArchVersion (ARMv8p4) then return TRUE;
    return (HasArchVersion (ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");
```

## Library pseudocode for shared/functions/extension/HaveFlagFormatExt

```
// HaveFlagFormatExt ()
// =====
// Returns TRUE if flag format conversion instructions implemented
// and FALSE otherwise

boolean HaveFlagFormatExt ()
    return HasArchVersion (ARMv8p5);
```



### Library pseudocode for shared/functions/extension/HaveFlagManipulateExt

```
// HaveFlagManipulateExt()
// =====
// Returns TRUE if has flag manipulate instructions, and FALSE otherwise

boolean HaveFlagManipulateExt()
    return HasArchVersion(ARMv8p4);
```

### Library pseudocode for shared/functions/extension/HaveFrintExt

```
// HaveFrintExt()
// =====
// Returns TRUE if FRINT instructions are implemented and FALSE otherwise

boolean HaveFrintExt()
    return HasArchVersion(ARMv8p5);
```

### Library pseudocode for shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// =====

boolean HaveHPMDExt()
    return HasArchVersion(ARMv8p1);
```

### Library pseudocode for shared/functions/extension/HaveIESB

```
// HaveIESB()
// =====

boolean HaveIESB()
    return (HaveRASExt() &&
           boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");
```

### Library pseudocode for shared/functions/extension/HaveMPAMExt

```
// HaveMPAMExt()
// =====
// Returns TRUE if MPAM implemented and FALSE otherwise.

boolean HaveMPAMExt()
    return (HasArchVersion(ARMv8p2) &&
           boolean IMPLEMENTATION_DEFINED "Has MPAM extension");
```

### Library pseudocode for shared/functions/extension/HaveMTEExt

```
// HaveMTEExt()
// =====
// Returns TRUE if MTE implemented and FALSE otherwise.

boolean HaveMTEExt()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE extension";
```

### Library pseudocode for shared/functions/extension/HaveNV2Ext

```
// HaveNV2Ext()
// =====

boolean HaveNV2Ext()
    return HasArchVersion(ARMv8p4);
```

### Library pseudocode for shared/functions/extension/HaveNVExt

```
// HaveNVExt()
// =====

boolean HaveNVExt()
    return HasArchVersion(ARMv8p3);
```

### Library pseudocode for shared/functions/extension/HaveNoSecurePMUDisableOverride

```
// HaveNoSecurePMUDisableOverride()
// =====

boolean HaveNoSecurePMUDisableOverride()
    return HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/HaveNoninvasiveDebugAuth

```
// HaveNoninvasiveDebugAuth()
// =====
// Returns FALSE if support for the removal of the non-invasive Debug controls is implemented.

boolean HaveNoninvasiveDebugAuth()
    return !HasArchVersion(ARMv8p4);
```

### Library pseudocode for shared/functions/extension/HavePANExt

```
// HavePANExt()
// =====

boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

### Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()
// =====

boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/HavePrivATExt

```
// HavePrivATExt()
// =====

boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt()
// =====

boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```

## Library pseudocode for shared/functions/extension/HaveRASExt

```
// HaveRASExt()
// =====

boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) ||
            boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

## Library pseudocode for shared/functions/extension/HaveSBExt

```
// HaveSBExt()
// =====
// Returns TRUE if has SB feature support, and FALSE otherwise.

boolean HaveSBExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension";
```

## Library pseudocode for shared/functions/extension/HaveSSBSExt

```
// HaveSSBSExt()
// =====
// Returns TRUE if has SSBS feature support, and FALSE otherwise.

boolean HaveSSBSExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension";
```

## Library pseudocode for shared/functions/extension/HaveSecureEL2Ext

```
// HaveSecureEL2Ext()
// =====
// Returns TRUE if has Secure EL2, and FALSE otherwise

boolean HaveSecureEL2Ext()
    return HasArchVersion(ARMv8p4);
```

## Library pseudocode for shared/functions/extension/HaveSecureExtDebugView

```
// HaveSecureExtDebugView()
// =====
// Returns TRUE if supports Secure and Non-secure views of debug peripherals is implemented.

boolean HaveSecureExtDebugView()
    return HasArchVersion(ARMv8p4);
```

## Library pseudocode for shared/functions/extension/HaveSelfHostedTrace

```
// HaveSelfHostedTrace()
// =====

boolean HaveSelfHostedTrace()
    return HasArchVersion(ARMv8p4);
```

## Library pseudocode for shared/functions/extension/HaveSmallPageTblExt

```
// HaveSmallPageTblExt()
// =====
// Returns TRUE if has Small Page Table Support, and FALSE otherwise

boolean HaveSmallPageTblExt()
    return HasArchVersion(ARMv8p4) && boolean IMPLEMENTATION_DEFINED "Has Small Page Table extension";
```

### Library pseudocode for shared/functions/extension/HaveStage2MemAttrControl

```
// HaveStage2MemAttrControl()
// =====
// Returns TRUE if support for Stage2 control of memory types and cacheability attributes is implemented
boolean HaveStage2MemAttrControl()
    return HasArchVersion(ARMv8p4);
```

### Library pseudocode for shared/functions/extension/HaveStatisticalProfiling

```
// HaveStatisticalProfiling()
// =====

boolean HaveStatisticalProfiling()
    return HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```
// HaveTrapLoadStoreMultipleDeviceExt()
// =====

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/HaveUA16Ext

```
// HaveUA16Ext()
// =====
// Returns TRUE if has extended unaligned memory access support, and FALSE otherwise

boolean HaveUA16Ext()
    return HasArchVersion(ARMv8p4);
```

### Library pseudocode for shared/functions/extension/HaveUAOExt

```
// HaveUAOExt()
// =====

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);
```

### Library pseudocode for shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt()
// =====

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
```

### Library pseudocode for shared/functions/extension/InsertIESBBeforeException

```
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.
boolean InsertIESBBeforeException(bits(2) el);
```

## Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// =====

// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRTType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding\_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0');
    else
        result = FPRound(real_operand, fpcr, rounding);

    return result;
```

## Library pseudocode for shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =====

bits(N) FPAbs(bits(N) op)
    assert N IN {16,32,64};
    return '0' : op<N-2:0>;
```

## Library pseudocode for shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType\_Infinity); inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero); zero2 = (type2 == FPType\_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

## Library pseudocode for shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// =====

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPTType\_SNaN || type1==FPTType\_QNaN || type2==FPTType\_SNaN || type2==FPTType\_QNaN then
        result = '0011';
        if type1==FPTType\_SNaN || type2==FPTType\_SNaN || signal_nans then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elseif value1 < value2 then
            result = '1000';
        else // value1 > value2
            result = '0010';
    return result;
```

## Library pseudocode for shared/functions/float/fpcompareeq/FPCompareEQ

```
// FPCompareEQ()
// =====

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPTType\_SNaN || type1==FPTType\_QNaN || type2==FPTType\_SNaN || type2==FPTType\_QNaN then
        result = FALSE;
        if type1==FPTType\_SNaN || type2==FPTType\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;
```

## Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

```
// FPCompareGE()
// =====

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPTType\_SNaN || type1==FPTType\_QNaN || type2==FPTType\_SNaN || type2==FPTType\_QNaN then
        result = FALSE;
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
    return result;
```

## Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =====

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPTType\_SNaN || type1==FPTType\_QNaN || type2==FPTType\_SNaN || type2==FPTType\_QNaN then
        result = FALSE;
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
    return result;
```

## Library pseudocode for shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// =====

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRTType fpcr, FPRounding rounding)
    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (type,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if type == FPTType\_SNaN || type == FPTType\_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
            if type == FPTType\_SNaN || alt_hp then
                FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif type == FPTType\_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
    elsif type == FPTType\_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCV(value, fpcr, rounding);
    return result;

// FPConvert()
// =====

bits(M) FPConvert(bits(N) op, FPCRTType fpcr)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

## Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// =====

// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```

## Library pseudocode for shared/functions/float/fpcrtype/FPCRTType

```
type FPCRTType;
```

## Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecoderM()
// =====

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecoderM(bits(2) rm)
    case rm of
        when '00' return FPRounding_TIEAWAY; // A
        when '01' return FPRounding_TIEEVEN; // N
        when '10' return FPRounding_POSINF; // P
        when '11' return FPRounding_NEGINF; // M
```

## Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// =====

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF; // P
        when '10' return FPRounding_NEGINF; // M
        when '11' return FPRounding_ZERO; // Z
```



## Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// =====

bits(N) FPDefaultNaN()
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    sign = '0';
    exp = Ones(E);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =====

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTType Infinity);
        inf2 = (type2 == FPTType Infinity);
        zero1 = (type1 == FPTType Zero);
        zero2 = (type2 == FPTType Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then FPProcessException(FPExc\_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);
    return result;
```

## Library pseudocode for shared/functions/float/fpexc/FPExc

```
enumeration FPExc
    {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
     FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

## Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```
// FPInfinity()
// =====

bits(N) FPInfinity(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Ones(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// =====

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 > value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType Infinity then
            result = FPInfinity(sign);
        elsif type == FPType Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            result = FPRound(value, fpcr);
    return result;
```

## Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =====

bits(N) FPMaxNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// =====

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // treat a single quiet-NaN as -Infinity
    if type1 == FPType QNaN && type2 != FPType QNaN then
        op1 = FPInfinity('1');
    elsif type1 != FPType QNaN && type2 == FPType QNaN then
        op2 = FPInfinity('1');

    return FPMax(op1, op2, fpcr);
```

## Library pseudocode for shared/functions/float/fpmin/FPMin

```
// FPMin()
// =====

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 < value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType Infinity then
            result = FPInfinity(sign);
        elsif type == FPType Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            result = FPRound(value, fpcr);
    return result;
```

## Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// =====

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // Treat a single quiet-NaN as +Infinity
    if type1 == FPType QNaN && type2 != FPType QNaN then
        op1 = FPInfinity('0');
    elsif type1 != FPType QNaN && type2 == FPType QNaN then
        op2 = FPInfinity('0');

    return FPMin(op1, op2, fpcr);
```

## Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// =====

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType Infinity);
        inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero);
        zero2 = (type2 == FPType Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

## Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd

```
// FPMulAdd()
// =====
//
// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPTYPE\_Infinity); zero1 = (type1 == FPTYPE\_Zero);
    inf2 = (type2 == FPTYPE\_Infinity); zero2 = (type2 == FPTYPE\_Zero);
    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);

    if typeA == FPTYPE\_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPTYPE\_Infinity); zeroA = (typeA == FPTYPE\_Zero);

        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
            result = FPDefaultNaN();
            FPProcessException(FPExc\_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elseif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elseif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elseif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH

```
// FPMulAddH()
// =====

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType Infinity); zero1 = (type1 == FPType Zero);
    inf2 = (type2 == FPType Infinity); zero2 = (type2 == FPType Zero);
    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);
    if typeA == FPType QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc InvalidOp, fpcr);
    if !done then
        infA = (typeA == FPType Infinity); zeroA = (typeA == FPType Zero);
        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;
        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
            result = FPDefaultNaN();
            FPProcessException(FPExc InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');
        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);
        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

## Library pseudocode for shared/functions/float/fpmuladdh/FPPProcessNaNs3H

```
// FPPProcessNaNs3H()
// =====

(boolean, bits(N)) FPPProcessNaNs3H(FPType type1, FPType type2, FPType type3, bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3)
assert N IN {32,64};
bits(N) result;
if type1 == FPType_SNaN then
    done = TRUE; result = FPPProcessNaN(type1, op1, fpcr);
elsif type2 == FPType_SNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type2, op2, fpcr));
elsif type3 == FPType_SNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type3, op3, fpcr));
elsif type1 == FPType_QNaN then
    done = TRUE; result = FPPProcessNaN(type1, op1, fpcr);
elsif type2 == FPType_QNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type2, op2, fpcr));
elsif type3 == FPType_QNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type3, op3, fpcr));
else
    done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);
```

## Library pseudocode for shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// =====

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRTYPE fpcr)
assert N IN {16,32,64};
bits(N) result;
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);
    if (inf1 && zero2) || (zero1 && inf2) then
        result = FPTwo(sign1 EOR sign2);
    elsif inf1 || inf2 then
        result = FPInfinity(sign1 EOR sign2);
    elsif zero1 || zero2 then
        result = FPZero(sign1 EOR sign2);
    else
        result = FPRound(value1*value2, fpcr);
return result;
```

## Library pseudocode for shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =====

bits(N) FPNeg(bits(N) op)
assert N IN {16,32,64};
return NOT(op<N-1>) : op<N-2:0>;
```

## Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// =====

bits(N) FPOnePointFive(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// =====
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRTYPE fpcr)
    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp      cumul = 0;
        when FPExc_DivideByZero    cumul = 1;
        when FPExc_Overflow        cumul = 2;
        when FPExc_Underflow       cumul = 3;
        when FPExc_Inexact         cumul = 4;
        when FPExc_InputDenorm     cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrapException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

## Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// =====

bits(N) FPProcessNaN(FPType type, bits(N) op, FPCRTYPE fpcr)
    assert N IN {16,32,64};
    assert type IN {FPType_QNaN, FPType_SNaN};

    case N of
        when 16 topfrac = 9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if type == FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then // DefaultNaN requested
        result = FPDefaultNaN();
    return result;
```



## Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                bits(N) op1, bits(N) op2,
                                FPCRTYPE fpcr)

assert N IN {16,32,64};
if type1 == FPType SNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elseif type2 == FPType SNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elseif type1 == FPType QNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elseif type2 == FPType QNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
else
    done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);
```

## Library pseudocode for shared/functions/float/fpprocessnans3/FPProcessNaNs3

```
// FPProcessNaNs3()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                bits(N) op1, bits(N) op2, bits(N) op3,
                                FPCRTYPE fpcr)

assert N IN {16,32,64};
if type1 == FPType SNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elseif type2 == FPType SNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elseif type3 == FPType SNaN then
    done = TRUE; result = FPProcessNaN(type3, op3, fpcr);
elseif type1 == FPType QNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elseif type2 == FPType QNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elseif type3 == FPType QNaN then
    done = TRUE; result = FPProcessNaN(type3, op3, fpcr);
else
    done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);
```



```

// FPrecepEstimate()
// =====

bits(N) FPrecepEstimate(bits(N) operand, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPType\_SNaN || type == FPType\_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPType\_Infinity then
        result = FPZero(sign);
    elsif type == FPType\_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc\_DivideByZero, fpcr);
    elsif (
        (N == 16 && Abs(value) < 2.0^-16) ||
        (N == 32 && Abs(value) < 2.0^-128) ||
        (N == 64 && Abs(value) < 2.0^-1024)
    ) then
        case FPRoundingMode(fpcr) of
            when FPRounding\_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding\_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding\_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding\_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign) else FPMMaxNormal(sign);
        FPProcessException(FPExc\_Overflow, fpcr);
        FPProcessException(FPExc\_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
        && (
            (N == 16 && Abs(value) >= 2.0^14) ||
            (N == 32 && Abs(value) >= 2.0^126) ||
            (N == 64 && Abs(value) >= 2.0^1022)
        ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign);
        if UsingAArch32() then
            FPSR.UFC = '1';
        else
            FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            if fraction<51> == 0 then
                exp = -1;
                fraction = fraction<49:0>:'00';
            else
                fraction = fraction<50:0>:'0';

        integer scaled = UInt('1':fraction<51:44>);

        case N of
            when 16 result_exp = 29 - exp; // In range 29-30 = -1 to 29+1 = 30
            when 32 result_exp = 253 - exp; // In range 253-254 = -1 to 253+1 = 254

```

```

        when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

// scaled is in range 256..511 representing a fixed-point number in range [0.5..1.0)
estimate = RecipEstimate(scaled);

// estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
// Convert to scaled floating point result with copied sign bit,
// high-order bits from estimate, and exponent calculated above.

fraction = estimate<7:0> : Zeros(44);
if result_exp == 0 then
    fraction = '1' : fraction<51:1>;
elsif result_exp == -1 then
    fraction = '01' : fraction<51:2>;
    result_exp = 0;

case N of
    when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
    when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
    when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

return result;

```

### Library pseudocode for shared/functions/float/fprecipestimate/RecipEstimate

```

// Compute estimate of reciprocal of 9-bit fixed-point number
//
// a is in range 256 .. 511 representing a number in the range 0.5 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipEstimate(integer a)
    assert 256 <= a && a < 512;
    a = a*2+1; // round to nearest
    integer b = (2 ^ 19) DIV a;
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;

```

## Library pseudocode for shared/functions/float/fprecpX/FPRecpX

```
// FPRecpX()
// =====

bits(N) FPRecpX(bits(N) op, FPCRTType fpcr)
    assert N IN {16,32,64};

    case N of
        when 16 esize = 5;
        when 32 esize = 8;
        when 64 esize = 11;

    bits(N)          result;
    bits(esize)      exp;
    bits(esize)      max_exp;
    bits(N-(esize+1)) frac = Zeros();

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp = Ones(esize) - 1;

    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPTType\_SNaN || type == FPTType\_QNaN then
        result = FPPProcessNaN(type, op, fpcr);
    else
        if IsZero(exp) then // Zero and denormals
            result = sign:max_exp:frac;
        else // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;
```



```

// FPRound()
// =====
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real op, FPCRTType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    return FPRoundBase(op, fpcr, rounding);

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRoundBase(real op, FPCRTType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding\_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Deal with flush-to-zero.
    if ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) && exponent < minimum_exp then
        // Flush-to-zero never generates a trapped exception
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            FPSR.UFC = '1';
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped.
    if biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
        FPProcessException(FPExc\_Underflow, fpcr);

    // Round result according to rounding mode.
    case rounding of
        when FPRounding\_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding\_POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding\_NEGINF
            round_up = (error != 0.0 && sign == '1');

```

```

        overflow_to_inf = (sign == '1');
    when FPRounding\_ZERO, FPRounding\_ODD
        round_up = FALSE;
        overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then          // Rounded up from denormalized to normalized
            biased_exp = 1;
            if int_mant == 2^(F+1) then    // Rounded up to next exponent
                biased_exp = biased_exp + 1;  int_mant = int_mant DIV 2;

// Handle rounding to odd aka Von Neumann rounding
if error != 0.0 && rounding == FPRounding\_ODD then
    int_mant<0> = '1';

// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then
        result = if overflow_to_inf then FPInfinity(sign) else FPMMaxNormal(sign);
        FPPProcessException(FPExc\_Overflow, fpcr);
        error = 1.0; // Ensure that an Inexact exception occurs
    else
        result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;
else // Alternative half precision
    if biased_exp >= 2^E then
        result = sign : Ones(N-1);
        FPPProcessException(FPExc\_InvalidOp, fpcr);
        error = 0.0; // Ensure that an Inexact exception does not occur
    else
        result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;

// Deal with Inexact exception.
if error != 0.0 then
    FPPProcessException(FPExc\_Inexact, fpcr);

return result;

// FPRound()
// =====

bits(N) FPRound(real op, FPCRTType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));

```

### Library pseudocode for shared/functions/float/fpround/FPRoundCV

```

// FPRoundCV()
// =====
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRTType fpcr, FPRounding rounding)
    fpcr.FZ16 = '0';
    return FPRoundBase(op, fpcr, rounding);

```

### Library pseudocode for shared/functions/float/fprounding/FPRounding

```

enumeration FPRounding {FPRounding\_TIEEVEN, FPRounding\_POSINF,
                        FPRounding\_NEGINF,  FPRounding\_ZERO,
                        FPRounding\_TIEAWAY, FPRounding\_ODD};

```



## Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

```
// FPRoundingMode()
// =====

// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRTYPE fpcr)
    return FPDecodeRounding(fpcr.RMode);
```

## Library pseudocode for shared/functions/float/fproundint/FPRoundInt

```
// FPRoundInt()
// =====

// Round OP to nearest integral floating point value using rounding mode ROUNDING.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to OP.

bits(N) FPRoundInt(bits(N) op, FPCRTYPE fpcr, FPRounding rounding, boolean exact)
    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    if type == FPTYPE_SNaN || type == FPTYPE_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPTYPE_Infinity then
        result = FPInfinity(sign);
    elsif type == FPTYPE_Zero then
        result = FPZero(sign);
    else
        // extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact
        if real_result == 0.0 then
            result = FPZero(sign);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO);

        // Generate inexact exceptions
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```



```

// FPRoundIntN()
// =====

bits(N) FPRoundIntN(bits(N) op, FPCRTType fpcr, FPRounding rounding, integer intsize)
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    if type IN {FPTType_SNaN, FPTType_QNaN, FPTType_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif type == FPTType_Zero then
            result = FPZero(sign);
        else
            // Extract integer component
            int_result = RoundDown(value);
            error = value - Real(int_result);

            // Determine whether supplied rounding mode requires an increment
            case rounding of
                when FPRounding_TIEEVEN
                    round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
                when FPRounding_POSINF
                    round_up = error != 0.0;
                when FPRounding_NEGINF
                    round_up = FALSE;
                when FPRounding_ZERO
                    round_up = error != 0.0 && int_result < 0;
                when FPRounding_TIEAWAY
                    round_up = error > 0.5 || (error == 0.5 && int_result >= 0);

            if round_up then int_result = int_result + 1;

            if int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1) then
                if N == 32 then
                    exp = 126 + intsize;
                    result = '1':exp<(E-1):0>:Zeros(F);
                else
                    exp = 1022 + intsize;
                    result = '1':exp<(E-1):0>:Zeros(F);
                    FPProcessException(FPExc_InvalidOp, fpcr);
                    // this case shouldn't set Inexact
                    error = 0.0;

            else
                // Convert integer value into an equivalent real value
                real_result = Real(int_result);

                // Re-encode as a floating-point value, result is always exact
                if real_result == 0.0 then
                    result = FPZero(sign);
                else
                    result = FPRound(real_result, fpcr, FPRounding_ZERO);

            // Generate inexact exceptions
            if error != 0.0 then
                FPProcessException(FPExc_Inexact, fpcr);

    return result;

```

## Library pseudocode for shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =====

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPTType\_SNaN || type == FPTType\_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPTType\_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc\_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif type == FPTType\_Infinity then
        result = FPZero('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == 0 do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('01':fraction<51:45>);

        case N of
            when 16 result_exp = ( 44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros( 2);
            when 32 result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
            when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);
    return result;
```

## Library pseudocode for shared/functions/float/fpsqrtestimate/RecipSqrtEstimate

```
// Compute estimate of reciprocal square root of 9-bit fixed-point number
//
// a is in range 128 .. 511 representing a number in the range 0.25 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range 1.0 to 511/256.

integer RecipSqrtEstimate(integer a)
    assert 128 <= a && a < 512;
    if a < 256 then // 0.25 .. 0.5
        a = a*2+1; // a in units of 1/512 rounded to nearest
    else // 0.5 .. 1.0
        a = (a >> 1) << 1; // discard bottom bit
        a = (a+1)*2; // a in units of 1/256 rounded to nearest
    integer b = 512;
    while a*(b+1)*(b+1) < 2^28 do
        b = b+1;
    // b = largest b such that b < 2^14 / sqrt(a) do
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

## Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// =====

bits(N) FPSqrt(bits(N) op, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPTType\_SNaN || type == FPTType\_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPTType\_Zero then
        result = FPZero(sign);
    elsif type == FPTType\_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr);
    return result;
```

## Library pseudocode for shared/functions/float/fpsub/FPSub

```
// FPSub()
// =====

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType Infinity);
        inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero);
        zero2 = (type2 == FPType Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

## Library pseudocode for shared/functions/float/fpthree/FPThree

```
// FPThree()
// =====

bits(N) FPThree(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// =====

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRTType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding\_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    // If NaN, set cumulative flag or take exception
    if type == FPType\_SNaN || type == FPType\_QNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment
    case rounding of
        when FPRounding\_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding\_POSINF
            round_up = (error != 0.0);
        when FPRounding\_NEGINF
            round_up = FALSE;
        when FPRounding\_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding\_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc\_Inexact, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

```
// FPToFixedJS()
// =====

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

bits(N) FPToFixedJS(bits(M) op, FPCRTType fpcr, boolean Is64)

    assert M == 64 && N == 32;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    Z = '1';
    // If NaN, set cumulative flag or take exception
    if type == FPTType\_SNaN || type == FPTType\_QNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);
        Z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc\_InvalidOp, fpcr);
        Z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc\_Inexact, fpcr);
        Z = '0';
    if sign == '1' && value == 0.0 then
        Z = '0';

    if type == FPTType\_Infinity then result = 0;

    if Is64 then
        PSTATE.<N,Z,C,V> = '0':Z:'00';
    else
        FPSCR<31:28> = '0':Z:'00';

    return result<N-1:0>;
```

## Library pseudocode for shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =====

bits(N) FPTwo(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```



## Library pseudocode for shared/functions/float/fptype/FPType

```
enumeration FPType      {FPType_Nonzero, FPType_Zero, FPType_Infinity,  
                          FPType_QNaN, FPType_SNaN};
```

## Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()  
// =====  
//  
// Used by data processing and int/fixed <-> FP conversion instructions.  
// For half-precision data it ignores AHP, and observes FZ16.  
  
(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)  
    fpcr.AHP = '0';  
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);  
    return (fp_type, sign, value);
```



```

// FPUnpackBase()
// =====
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRTType fpcr)
    assert N IN {16,32,64};

    if N == 16 then
        sign    = fpval<15>;
        exp16   = fpval<14:10>;
        frac16  = fpval<9:0>;
        if IsZero(exp16) then
            // Produce zero if value is zero or flush-to-zero is selected
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                type = FPType\_Zero; value = 0.0;
            else
                type = FPType\_Nonzero; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                type = FPType\_Infinity; value = 2.0^1000000;
            else
                type = if frac16<9> == '1' then FPType\_QNaN else FPType\_SNaN;
                value = 0.0;
        else
            type = FPType\_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 then

        sign    = fpval<31>;
        exp32   = fpval<30:23>;
        frac32  = fpval<22:0>;
        if IsZero(exp32) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac32) || fpcr.FZ == '1' then
                type = FPType\_Zero; value = 0.0;
                if !IsZero(frac32) then // Denormalized input flushed to zero
                    FPProcessException(FPExc\_InputDenorm, fpcr);
            else
                type = FPType\_Nonzero; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                type = FPType\_Infinity; value = 2.0^1000000;
            else
                type = if frac32<22> == '1' then FPType\_QNaN else FPType\_SNaN;
                value = 0.0;
        else
            type = FPType\_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64

        sign    = fpval<63>;
        exp64   = fpval<62:52>;
        frac64  = fpval<51:0>;
        if IsZero(exp64) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac64) || fpcr.FZ == '1' then
                type = FPType\_Zero; value = 0.0;
                if !IsZero(frac64) then // Denormalized input flushed to zero
                    FPProcessException(FPExc\_InputDenorm, fpcr);
            else

```

```

        type = FPTYPE\_Nonzero; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
    elsif IsOnes(exp64) then
        if IsZero(frac64) then
            type = FPTYPE\_Infinity; value = 2.0^1000000;
        else
            type = if frac64<51> == '1' then FPTYPE\_QNaN else FPTYPE\_SNaN;
            value = 0.0;
        else
            type = FPTYPE\_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;
    return (type, sign, value);

```

### Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

```

// FPUnpackCV()
// =====
//
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPTYPE, bit, real) FPUnpackCV(bits(N) fpval, FPCRTYPE fpcr)
    fpcr.FZ16 = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);

```

### Library pseudocode for shared/functions/float/fpzero/FPZero

```

// FPZero()
// =====

bits(N) FPZero(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Zeros(E);
    frac = Zeros(F);
    return sign : exp : frac;

```

### Library pseudocode for shared/functions/float/vfpexpandimm/VFPEExpandImm

```

// VFPEExpandImm()
// =====

bits(N) VFPEExpandImm(bits(8) imm8)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    return sign : exp : frac;

```

## Library pseudocode for shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

## Library pseudocode for shared/functions/memory/AArch64.BranchAddr

```
// AArch64.BranchAddr()
// =====
// Return the virtual address with tag bits removed for storing to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, PSTATE.EL);
    if msbit == 63 then
        return vaddress;
    elsif (PSTATE.EL IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>);
    else
        return ZeroExtend(vaddress<msbit:0>);
```

## Library pseudocode for shared/functions/memory/AccType

```
enumeration AccType {AccType_NORMAL, AccType_VEC, // Normal loads and stores
    AccType_STREAM, AccType_VECSTREAM, // Streaming loads and stores
    AccType_ATOMIC, AccType_ATOMICRW, // Atomic loads and stores
    AccType_ORDERED, AccType_ORDEREDRW, // Load-Acquire and Store-Release
    AccType_ORDEREDATOMIC, // Load-Acquire and Store-Release with atomic
    AccType_ORDEREDATOMICRW, AccType_UNPRIV, // Load and store unpr
    AccType_LIMITEDORDERED, // Load-LOAcquire and Store-LOReleaseAccType_I
    AccType_UNPRIV, // Load and store unprivilegedAccType_PTW,
    AccType_IFETCH, // Instruction fetchAccType_NONFAULT,
    AccType_PTW, // Page table walkAccType_CNOTFIRST,
    AccType_NONFAULT, // Non-faulting loadsAccType_NV2REGISTER,
    // to a memory access that uses the EL2 transl
    // Other operations
    AccType_CNOTFIRST, // Contiguous FF load, not first elementAccType
    AccType_NV2REGISTER, // MRS/MSR instruction used at EL1 and which i
    // to a memory access that uses the EL2 transl
    // Other operationsAccType_IC, // Instruction cache mainte
    AccType_DC, // Data cache maintenanceAccType_DCZVA,
    AccType_DC_UNPRIV, // Data cache maintenance instruction used at
    AccType_IC, // Instruction cache maintenance
    AccType_DCZVA, // DC ZVA instructions
    AccType_AT}; // Address translation
```

## Library pseudocode for shared/functions/memory/AccessDescriptor

```
type AccessDescriptor is (type
    AccessDescriptor is {
    AccType acctype,
    MPAMInfo mpam,
    boolean page_table_walk,
    boolean secondstage,
    boolean s2fslwalk,
    integer level
    )
```

## Library pseudocode for shared/functions/memory/AddrTop

```
// AddrTop()
// =====
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        // AArch64 translation regime.
        case regime of
            when EL1
                tbi = (if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0);
                if HavePACExt() then
                    tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
            when EL2
                if HaveVirtHostExt() && ELIsInHost(el) then
                    tbi = (if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0);
                    if HavePACExt() then
                        tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
                else
                    tbi = TCR_EL2.TBI;
                    if HavePACExt() then tbid = TCR_EL2.TBID;
            when EL3
                tbi = TCR_EL3.TBI;
                if HavePACExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr) then 55 else 63);
```

## Library pseudocode for shared/functions/memory/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord      fault,      // fault.type indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    bits(64)         vaddress
)
```

## Library pseudocode for shared/functions/memory/AddressWithAllocationTag

```
// AddressWithAllocationTag()
// =====
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as "0000".

bits(64) AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag = allocation_tag - ('000':address<55>);
    result<59:56> = tag;
    return result;
```

## Library pseudocode for shared/functions/memory/Allocation

```
constant bits(2) MemHint_No = '00';    // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01';    // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10';    // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';   // Read-Allocate, Write-Allocate
```

## Library pseudocode for shared/functions/memory/AllocationTagFromAddress

```
// AllocationTagFromAddress()
// =====
// Generate a Tag from a 64-bit value containing a Logical Address Tag.
// If access to Allocation Tags is disabled, this function returns â€™0000â€™.

bits(4) AllocationTagFromAddress(bits(64) tagged_address)
    bits(4) logical_tag = tagged_address<59:56>;
    bits(4) tag = logical_tag + ('000':tagged_address<55>);
    return tag;
```

## Library pseudocode for shared/functions/memory/BigEndian

```
// BigEndian()
// =====

boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].EOE != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

## Library pseudocode for shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// =====

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```

## Library pseudocode for shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00'; // Non-cacheable
constant bits(2) MemAttr_WT = '10'; // Write-through
constant bits(2) MemAttr_WB = '11'; // Write-back
```

## Library pseudocode for shared/functions/memory/CheckTagCreateAccessDescriptor

```
// CheckTag()
// =====
// Performs a Tag Check operation for a memory access and returns
// whether the check passed
// CreateAccessDescriptor()
// =====

boolean AccessDescriptor CheckTag(CreateAccessDescriptor(AddressDescriptor AccType memaddrdesc, bits(4) p
    if memaddrdesc.memattrs.tagged then
        bits(64) paddress = acctype ZeroExtendAccessDescriptor(memaddrdesc.paddress.address);
        return ptag == MemTag[paddress];
    else
        return TRUE; accdesc;
    accdesc.acctype = acctype;
    accdesc.page_table_walk = FALSE;
    return accdesc;
```

## Library pseudocode for shared/functions/memory/AccessDescriptorCreateAccessDescriptorPTW

```
// CreateAccessDescriptor()
// =====
// CreateAccessDescriptorPTW()
// =====

AccessDescriptor CreateAccessDescriptor(AccessDescriptorCreateAccessDescriptorPTW(AccType acctype)
AccessDescriptor accdesc;
accdesc.acctype = acctype;
accdesc.mpam = GenMPAMcurEL(acctype IN {acctype, boolean secondstage,
boolean s2fslwalk, integer level}) AccType IFETCH AccessDescriptor
accdesc.page_table_walk = FALSE;
accdesc;
accdesc.acctype = acctype;
accdesc.page_table_walk = TRUE;
accdesc.secondstage = s2fslwalk;
accdesc.secondstage = secondstage;
accdesc.level = level;
return accdesc;
```

## Library pseudocode for shared/functions/memory/AccessDescriptorPTW

```
// CreateAccessDescriptorPTW()
// =====

AccessDescriptor CreateAccessDescriptorPTW(AccType acctype, boolean secondstage,
boolean s2fslwalk, integer level)
AccessDescriptor accdesc;
accdesc.acctype = acctype;
accdesc.mpam = GenMPAMcurEL(acctype IN {AccType IFETCH, AccType IC});
accdesc.page_table_walk = TRUE;
accdesc.secondstage = s2fslwalk;
accdesc.secondstage = secondstage;
accdesc.level = level;
return accdesc;
```

## Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

## Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);
```

## Library pseudocode for shared/functions/memory/DescriptorUpdate

```
type DescriptorUpdate is (
    boolean AF, // AF needs to be set
    boolean AP, // AP[2] / S2AP[2] will be modified
    AddressDescriptor descaddr // Descriptor to be updated
)
```

## Library pseudocode for shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```



## Library pseudocode for shared/functions/memory/EffectiveTBIFault

```
// EffectiveTBI()
// =====
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bitenumeration EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    assertFault { HaveEL(el);
    regime = Fault_None, S1TranslationRegime(el);
    assert(!Fault_AccessFlag, ELUsingAArch32(regime));

    case regime of
        when Fault_Alignment, EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if Fault_Background, HavePACEExt() then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        when Fault_Domain, EL2
            if Fault_Permission, HaveVirtHostExt() && Fault_Translation, ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                if Fault_AddressSize, HavePACEExt() then
                    tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
            else
                tbi = TCR_EL2.TBI;
                if Fault_SyncExternal, HavePACEExt() then tbid = TCR_EL2.TBID;
        when Fault_SyncExternalOnWalk, EL3
            tbi = TCR_EL3.TBI;
            if Fault_SyncParity, HavePACEExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!Fault_SyncParityOnWalk, Fault_AsyncParity,
        Fault_AsyncExternal,
        Fault_Debug,
        Fault_TLBConflict,
        Fault_HWUpdateAccessFlag,
        Fault_Lockdown,
        Fault_Exclusive,
        HavePACEExt() || tbid == '0' || !IsInstr) then '1' else '0'); Fault_ICacheMaint};
```

## Library pseudocode for shared/functions/memory/EffectiveTCMAFaultRecord

```
// EffectiveTCMA()
// =====
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bittype EffectiveTCMA(bits(64) address, bits(2) el)
    assertFaultRecord is ( HaveELFault(el);
    regime = type, // Fault Status S1TranslationRegimeAccType(el);
    assert(!acctype, // Type of access that faulted ELUsingAArch32FullAddress(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;

    return tcma; ipaddress, // Intermediate physical address
        boolean s2fslwalk, // Is on a Stage 1 page table walk
        boolean write, // TRUE for a write, FALSE for a read
        integer level, // For translation, access flag and permission faults
        bit extflag, // IMPLEMENTATION DEFINED syndrome for external aborts
        boolean secondstage, // Is a Stage 2 abort
        bits(4) domain, // Domain number, AArch32 only
        bits(2) errortype, // [ARMv8.2 RAS] AArch32 AET or AArch64 SET
        bits(4) debugmoe) // Debug method of entry, from AArch32 only
```

## Library pseudocode for shared/functions/memory/Fault

```
enumeration Fault {Fault_None,  
    Fault_AccessFlag,  
    Fault_Alignment,  
    Fault_Background,  
    Fault_Domain,  
    Fault_Permission,  
    Fault_Translation,  
    Fault_AddressSize,  
    Fault_SyncExternal,  
    Fault_SyncExternalOnWalk,  
    Fault_SyncParity,  
    Fault_SyncParityOnWalk,  
    Fault_AsyncParity,  
    Fault_AsyncExternal,  
    Fault_Debug,  
    Fault_TLBConflict,  
    Fault_BranchTarget,  
    Fault_HWUpdateAccessFlag,  
    Fault_Lockdown,  
    Fault_Exclusive,  
    Fault_ICacheMaint};
```

## Library pseudocode for shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault    type,          // Fault Status  
    AccType acctype,          // Type of access that faulted  
    FullAddress ipaddress,    // Intermediate physical address  
    boolean s2fslwalk,       // Is on a Stage 1 page table walk  
    boolean write,           // TRUE for a write, FALSE for a read  
    integer level,           // For translation, access flag and permission faults  
    bit extflag,             // IMPLEMENTATION DEFINED syndrome for external aborts  
    boolean secondstage,     // Is a Stage 2 abort  
    bits(4) domain,          // Domain number, AArch32 only  
    bits(2) errorrtype,      // [ARMv8.2 RAS] AArch32 AET or AArch64 SET  
    bits(4) debugmoe)        // Debug method of entry, from AArch32 only  
  
type PARTIDtype = bits(16);  
type PMGtype = bits(8);  
  
type MPAMinfo is (  
    bit mpam_ns,  
    PARTIDtype partid,  
    PMGtype pmg  
)
```

## Library pseudocode for shared/functions/memory/FullAddress

```
type FullAddress is (  
    bits(52) address,  
    bit      NS              // '0' = Secure, '1' = Non-secure  
)
```

## Library pseudocode for shared/functions/memory/Hint\_Prefetch

```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

## Library pseudocode for shared/functions/memory/MBReqDomain

```
enumeration MBReqDomain      {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                              MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

## Library pseudocode for shared/functions/memory/MBReqTypes

```
enumeration MBReqTypes      {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

## Library pseudocode for shared/functions/memory/MemAttrHints

```
type MemAttrHints is (
    bits(2) attrs, // See MemAttr_*, Cacheability attributes
    bits(2) hints, // See MemHint_*, Allocation hints
    boolean transient
)
```

## Library pseudocode for shared/functions/memory/MemTag

```
// MemTag[] - non-assignment (read) form
// =====
// Load an Allocation Tag from memory.

bits(4) MemTag[bits(64) address]
  AddressDescriptor memaddrdesc;
  bits(4) value;
  iswrite = FALSE;

memaddrdesc = AArch64.TranslateAddress(address, AccType NORMAL, iswrite, TRUE, TAG_GRANULE);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch64.Abort(address, memaddrdesc.fault);

// Return the granule tag if tagging is enabled...
if AllocationTagAccessIsEnabled() then
  return MemTag[memaddrdesc];
else
  // ...otherwise read tag as zero.
  return '0000';

// MemTag[] - assignment (write) form
// =====
// Store an Allocation Tag to memory.

MemTag[bits(64) address] = bits(4) value
  AddressDescriptor memaddrdesc;
  iswrite = TRUE;

// Stores of allocation tags must be aligned
if address != Align(address, TAG_GRANULE) then
  boolean secondstage = FALSE;
  AArch64.Abort(address, AArch64.AlignmentFault(AccType NORMAL, iswrite, secondstage));

wasaligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, AccType NORMAL, iswrite, wasaligned, TAG_GRANULE);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
if AllocationTagAccessIsEnabled() then
  _MemTag[memaddrdesc] = value;
```

## Library pseudocode for shared/functions/memory/MemType

```
enumeration MemType {MemType_Normal, MemType_Device};
```

## Library pseudocode for shared/functions/memory/MemoryAttributes

```
type MemoryAttributes is (
  MemType      type,

  DeviceType   device,      // For Device memory types
  MemAttrHints inner,      // Inner hints and attributes
  MemAttrHints outer,      // Outer hints and attributes
  boolean      tagged,     // Tagged access

  boolean      shareable,
  boolean      outershareable
)
```

## Library pseudocode for shared/functions/memory/Permissions

```
type Permissions is (  
    bits(3) ap,    // Access permission bits  
    bit      xn,    // Execute-never bit  
    bit      xxn,   // [ARMv8.2] Extended execute-never bit for stage 2  
    bit      pxn    // Privileged execute-never bit  
)
```

## Library pseudocode for shared/functions/memory/PrefetchHint

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

## Library pseudocode for shared/functions/memory/SpeculativeSynchronizationBarrierToPA

```
SpeculativeSynchronizationBarrierToPA();
```

## Library pseudocode for shared/functions/memory/SpeculativeSynchronizationBarrierToVA

```
SpeculativeSynchronizationBarrierToVA();
```

## Library pseudocode for shared/functions/memory/TLBRecord

```
type TLBRecord is (  
    Permissions      perms,  
    bit              nG,                // '0' = Global, '1' = not Global  
    bits(4)          domain,           // AArch32 only  
    bit              GP,               // Guarded Page  
    boolean          contiguous,       // Contiguous bit from page table  
    integer          level,            // AArch32 Short-descriptor format: Indicates Section/Page  
    integer          blocksize,        // Describes size of memory translated in KBytes  
    DescriptorUpdate descupdate,       // [ARMv8.1] Context for h/w update of table descriptor  
    bit              CnP,              // [ARMv8.2] TLB entry can be shared between different PEs  
    AddressDescriptor addrdesc  
)
```

## Library pseudocode for shared/functions/memory/TransformTag

```
// TransformTag()  
// =====  
// Apply tag transformation rules.  
  
bits(4) TransformTag(bits(64) vaddr)  
    bits(4) vtag = vaddr<59:56>;  
    bits(4) tagdelta = ZeroExtend(vaddr<55>);  
    bits(4) ptag = vtag + tagdelta;  
    return ptag;
```

## Library pseudocode for shared/functions/memory/\_Mem

```
// These two _Mem[] accessors are the hardware operations which perform single-copy atomic,  
// aligned, little-endian memory accesses of size bytes from/to the underlying physical  
// memory array of bytes.  
//  
// The functions address the array using desc.paddress which supplies:  
// * A 52-bit physical address  
// * A single NS bit to select between Secure and Non-secure parts of the array.  
//  
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,  
// etc and other parameters required to access the physical memory or for setting syndrome  
// register in the event of an external abort.  
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc];  
  
_Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc] = bits(8*size) value;
```

## Library pseudocode for shared/functions/memoryregisters/booleanBranchTo

```
// boolean AccessIsTagChecked()
// =====
// TRUE if a given access is tag-checked, FALSE otherwise.
// BranchTo()
// =====

boolean// Set program counter to a new address, which might include a tag in the top eight bits,
// with a branch reason hint for possible use by hardware fetching the next instruction. AccessIsTagChe
    if PSTATE.M<4> == '1' then return FALSE;

    if(branch type) EffectiveTBIHint_Branch(vaddr, FALSE, PSTATE.EL) == '0' then
        return FALSE;

    if(branch type);
    if N == 32 then
        assert EffectiveTCMAUsingAArch32(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<
        return FALSE;

    if !();
    PC = AllocationTagAccessIsEnabledZeroExtend() then
        return FALSE;

    if acctype IN {(target)};
    else
        assert N == 64 && !AccType_IFETCHUsingAArch32, ();
        PC = AccType_PTWAArch64.BranchAddr} then
            return FALSE;

    if acctype == AccType_NV2REGISTER then
        return FALSE;

    if PSTATE.TCO=='1' then
        return FALSE;

    if IsNonTagCheckedInstruction() then
        return FALSE;

    return TRUE;{(target<63:0>)};
return;
```

## Library pseudocode for shared/functions/mpamregisters/DefaultMPAMInfoBranchToAddr

```
// DefaultMPAMInfo
// =====
// Returns default MPAM info. If secure is TRUE return default Secure
// MPAMInfo, otherwise return default Non-secure MPAMInfo.
// BranchToAddr()
// =====

MPAMInfo DefaultMPAMInfo(boolean secure)
    MPAMInfo DefaultInfo;
    DefaultInfo.mpam_ns = if secure then '0' else '1';
    DefaultInfo.partid = DefaultPARTID;
    DefaultInfo.pmg = DefaultPMG;
    return DefaultInfo;

// Set program counter to a new address, which does not include a tag in the top
// with a branch reason hint for possible use by hardware fetching the next instruction.BranchToAddr(bi
    Hint_Branch(branch type);
    if N == 32 then
        assert UsingAArch32();
        PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        PC = target<63:0>;
    return;
```

## Library pseudocode for shared/functions/mpam/registers/DefaultPARTIDBranchType

```
constant PARTIDtype DefaultPARTID = 0<15:0>; enumeration BranchType {BranchType_CALL, BranchType_ERET, Br
    BranchType_RET, BranchType_JMP, BranchType_EXCEPTION,
    BranchType_UNKNOWN};
```

## Library pseudocode for shared/functions/mpam/DefaultPMG

```
constant PMGtype DefaultPMG = 0<7:0>;
```

## Library pseudocode for shared/functions/mpam/GenMPAMcurEL

```
// GenMPAMcurEL
// =====
// Returns MPAMinfo for the current EL and security state.
// InD is TRUE instruction access and FALSE otherwise.
// May be called if MPAM is not implemented (but in a version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(boolean InD)
    bits(2) mpamel;
    boolean validEL;
    boolean secure = IsSecure();
    if HaveMPAMExt() && MPAMisEnabled() then
        if UsingAArch32() then
            (validEL, mpamel) = ELFromM32(PSTATE.M);
        else
            validEL = TRUE;
            mpamel = PSTATE.EL;
        if validEL then
            return genMPAM(UInt(mpamel), InD, secure);
    return DefaultMPAMinfo(secure);
```

## Library pseudocode for shared/functions/mpam/MAP\_vPARTID

```
// MAP_vPARTID
// =====
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    PARTIDtype ret;
    boolean err;
    integer virt      = UInt( vpartid );
    integer vmprmax = UInt( MPAMIDR_EL1.VPMR_MAX );

    // vpartid_max is largest vpartid supported
    integer vpartid_max = 4 * vmprmax + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if virt > vpartid_max then
        virt = virt MOD (vpartid_max+1);

    // Check for valid mapping entry.
    if MPAMVPMV_EL2<virt> == '1' then
        // vpartid has a valid mapping so access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Is the default virtual PARTID valid?
    elsif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
    else
        ret = DefaultPARTID;
        err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    integer partid_max = UInt( MPAMIDR_EL1.PARTID_MAX );
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DefaultPARTID;
        err = TRUE;

    return (ret, err);
```

## Library pseudocode for shared/functions/mpam/MPAMisEnabled

```
// MPAMisEnabled
// =====
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
    el = HighestEL();
    case el of
        when EL3 return MPAM3_EL3.MPAMEN == '1';
        when EL2 return MPAM2_EL2.MPAMEN == '1';
        when EL1 return MPAM1_EL1.MPAMEN == '1';
```



### Library pseudocode for shared/functions/mpam/MPAMisVirtual

```
// MPAMisVirtual
// =====
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(integer el)
    return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
            (( el == 0 && MPAMHCR_EL2.EL0_VPMEN == '1' ) ||
             ( el == 1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
```

### Library pseudocode for shared/functions/mpam/genMPAM

```
// genMPAM
// =====
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMEL_ELx register and otherwise using PARTID_D and PMG_D fields.
// Produces a Secure PARTID if Secure is TRUE and a Non-secure PARTID otherwise.

MPAMinfo genMPAM(integer el, boolean InD, boolean secure)
    MPAMinfo returnInfo;
    PARTIDtype partidel;
    boolean perr;
    boolean gstplk = (el == 0 && EL2Enabled() &&
                     MPAMHCR_EL2.GSTAPP_PLK == '1' && HCR_EL2.TGE == '0');
    integer eff_el = if gstplk then 1 else el;
    (partidel, perr) = genPARTID(eff_el, InD);
    PMGtype groupel = genPMG(eff_el, InD, perr);
    returnInfo.mpam_ns = if secure then '0' else '1';
    returnInfo.partid = partidel;
    returnInfo.pmg = groupel;
    return returnInfo;
```

### Library pseudocode for shared/functions/mpam/genMPAMEl

```
// genMPAMEl
// =====
// Returns MPAMinfo for specified EL in the current security state.
// InD is TRUE for instruction access and FALSE otherwise.

MPAMinfo genMPAMEl(bits(2) el, boolean InD)
    boolean secure = IsSecure();
    if HaveMPAMExt() && MPAMisEnabled() then
        return genMPAM(UInt(el), InD, secure);
    return DefaultMPAMinfo(secure);
```

### Library pseudocode for shared/functions/mpam/genPARTID

```
// genPARTID
// =====
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMEL_ELx.PARTID_I and
// otherwise from MPAMEL_ELx.PARTID_D.

(PARTIDtype, boolean) genPARTID(integer el, boolean InD)
    PARTIDtype partidel = getMPAM_PARTID(el, InD);

    integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(partidel) > partid_max then
        return (DefaultPARTID, TRUE);

    if MPAMisVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```

## Library pseudocode for shared/functions/mpam/genPMG

```
// genPMG
// =====
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(integer el, boolean InD, boolean partid_err)
    integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);

    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DefaultPMG;
    PMGtype groupel = getMPAM_PMG(el, InD);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DefaultPMG;
```

## Library pseudocode for shared/functions/mpam/getMPAM\_PARTID

```
// getMPAM_PARTID
// =====
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(integer MPAMn, boolean InD)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when 3 partid = MPAM3_EL3.PARTID_I;
            when 2 partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros();
            when 1 partid = MPAM1_EL1.PARTID_I;
            when 0 partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
    else
        case MPAMn of
            when 3 partid = MPAM3_EL3.PARTID_D;
            when 2 partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros();
            when 1 partid = MPAM1_EL1.PARTID_D;
            when 0 partid = MPAM0_EL1.PARTID_D;
            otherwise partid = PARTIDtype UNKNOWN;
    return partid;
```

## Library pseudocode for shared/functions/mpam/getMPAM\_PMG

```
// getMPAM_PMG
// =====
// Returns a PMG from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(integer MPAMn, boolean InD)
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when 3 pmg = MPAM3_EL3.PMG_I;
            when 2 pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros();
            when 1 pmg = MPAM1_EL1.PMG_I;
            when 0 pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
        else
            case MPAMn of
                when 3 pmg = MPAM3_EL3.PMG_D;
                when 2 pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros();
                when 1 pmg = MPAM1_EL1.PMG_D;
                when 0 pmg = MPAM0_EL1.PMG_D;
                otherwise pmg = PMGtype UNKNOWN;
            return pmg;
```

## Library pseudocode for shared/functions/mpam/mapvpmw

```
// mapvpmw
// =====
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros(64);
    // vpme_lsb selects LSB of field within register
    integer vpme_lsb = (vpartid REM 4) * 16;
    return vpmw<vpme_lsb +: 16>;
```

## Library pseudocode for shared/functions/registers/BranchTo

```
// BranchTo()
// =====

// Set program counter to a new address, with a branch type
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = AArch64.BranchAddr(target<63:0>);
    return;
```

## Library pseudocode for shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// =====

// Set program counter to a new address, with a branch type
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

## Library pseudocode for shared/functions/registers/BranchType

```
enumeration BranchType {
    BranchType_DIRCALL,    // Direct Branch with link
    BranchType_INDICAL,    // Indirect Branch with link
    BranchType_ERET,       // Exception return (indirect)
    BranchType_DBGEXIT,    // Exit from Debug state
    BranchType_RET,        // Indirect branch with function return hint
    BranchType_DIR,        // Direct branch
    BranchType_INDIR,       // Indirect branch
    BranchType_EXCEPTION,  // Exception entry
    BranchType_RESET,      // Reset
    BranchType_UNKNOWN};   // Other
```

## Library pseudocode for shared/functions/registers/Hint\_Branch

```
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.
Hint_Branch(BranchType hint);
```

## Library pseudocode for shared/functions/registers/NextInstrAddr

```
// Return address of the sequentially next instruction.
// Return address of the next instruction.
bits(N) NextInstrAddr();
```

## Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

```
// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);
```

### Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// =====
// Return address of the current instruction.

bits(N) ThisInstrAddr()
    assert N == 64 || (N == 32 && UsingAArch32\(\));
    return _PC<N-1:0>;
```

### Library pseudocode for shared/functions/registers/\_PC

```
bits(64) _PC;
```

### Library pseudocode for shared/functions/registers/\_R

```
array bits(64) _R[0..30];
```

### Library pseudocode for shared/functions/registers/\_V

```
array bits(128) _V[0..31];
```

## Library pseudocode for shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// =====

bits(32) SPSR[]
  bits(32) result;
  if UsingAArch32\(\) then
    case PSTATE.M of
      when M32\_FIQ      result = SPSR_fiq;
      when M32\_IRQ      result = SPSR_irq;
      when M32\_Svc       result = SPSR_svc;
      when M32\_Monitor   result = SPSR_mon;
      when M32\_Abort     result = SPSR_abt;
      when M32\_Hyp       result = SPSR_hyp;
      when M32\_Undef     result = SPSR_und;
      otherwise         Unreachable\(\);
  else
    case PSTATE.EL of
      when EL1           result = SPSR_EL1;
      when EL2           result = SPSR_EL2;
      when EL3           result = SPSR_EL3;
      otherwise         Unreachable\(\);

  return result;

// SPSR[] - assignment form
// =====

SPSR[] = bits(32) value
  if UsingAArch32\(\) then
    case PSTATE.M of
      when M32\_FIQ      SPSR_fiq = value;
      when M32\_IRQ      SPSR_irq = value;
      when M32\_Svc       SPSR_svc = value;
      when M32\_Monitor   SPSR_mon = value;
      when M32\_Abort     SPSR_abt = value;
      when M32\_Hyp       SPSR_hyp = value;
      when M32\_Undef     SPSR_und = value;
      otherwise         Unreachable\(\);
  else
    case PSTATE.EL of
      when EL1           SPSR_EL1 = value;
      when EL2           SPSR_EL2 = value;
      when EL3           SPSR_EL3 = value;
      otherwise         Unreachable\(\);

  return;
```

## Library pseudocode for shared/functions/system/AllocationTagAccessIsEnabledArchVersion

```
// AllocationTagAccessIsEnabled()
// =====
// Check whether access to Allocation Tags is enabled.

boolean enumeration AllocationTagAccessIsEnabled()
    if SCR_EL3.ATA == '0' && PSTATE.EL IN {ArchVersion-{EL0, ARMv8p0
, EL1, ARMv8p1
, EL2} then
        return FALSE;
    elsif HCR_EL2.ATA == '0' && HCR_EL2.<E2H,TGE> != '11' && PSTATE.EL IN {ARMv8p2
, EL0, ARMv8p3
, EL1} then
        return FALSE;
    elsif SCTLR_EL3.ATA == '0' && PSTATE.EL == EL3 then
        return FALSE;
    elsif SCTLR_EL2.ATA == '0' && PSTATE.EL == EL2 then
        return FALSE;
    elsif SCTLR_EL1.ATA == '0' && PSTATE.EL == EL1 then
        return FALSE;
    elsif SCTLR_EL2.ATA0 == '0' && HCR_EL2.<E2H,TGE> == '11' && PSTATE.EL == EL0 then
        return FALSE;
    elsif SCTLR_EL1.ATA0 == '0' && HCR_EL2.<E2H,TGE> != '11' && PSTATE.EL == EL0 then
        return FALSE;
    else
        return TRUE; ARMv8p4
};
```

## Library pseudocode for shared/functions/system/ArchVersion

```
enumeration ArchVersion {
    ARMv8p0
    , ARMv8p1
    , ARMv8p2
    , ARMv8p3
    , ARMv8p4
    , ARMv8p5
};
```

## Library pseudocode for shared/functions/system/BranchTargetCheck

```
// BranchTargetCheck()
// =====
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

BranchTargetCheck()

    assert HaveBTIExt() && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
        bits(64) pc = ThisInstrAddr();
        AArch64.BranchTargetException(pc<51:0>);

    boolean branch_instr = AArch64.ExecutingBROrBLROrRetInstr();
    boolean bti_instr = AArch64.ExecutingBTIInstr();

    // PSTATE.BTYPE defaults to 00 for instructions that don't explicitly set BTYPE.
    if !(branch_instr || bti_instr) then
        BTypeNext = '00';
```

### Library pseudocode for shared/functions/system/ChooseNonExcludedTag

```
// ChooseNonExcludedTag()
// =====
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) ChooseNonExcludedTag(bits(4) tag, bits(4) offset, bits(16) exclude)

    if exclude == Ones(16) then
        return tag;

    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    return tag;
```

### Library pseudocode for shared/functions/system/ClearEventRegister

```
// ClearEventRegister()
// =====
// Clear the Event Register of this PE

ClearEventRegister()
    EventRegister = '0';
    return;
```

### Library pseudocode for shared/functions/system/ClearPendingPhysicalSError

```
// Clear a pending physical SError interrupt
ClearPendingPhysicalSError();
```

### Library pseudocode for shared/functions/system/ClearPendingVirtualSError

```
// Clear a pending virtual SError interrupt
ClearPendingVirtualSError();
```

### Library pseudocode for shared/functions/system/ConditionHolds

```
// ConditionHolds()
// =====

// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1'); // EQ or NE
        when '001' result = (PSTATE.C == '1'); // CS or CC
        when '010' result = (PSTATE.N == '1'); // MI or PL
        when '011' result = (PSTATE.V == '1'); // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V); // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
        when '111' result = TRUE; // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```



## Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrier

```
ConsumptionOfSpeculativeDataBarrier();
```

## Library pseudocode for shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// =====

InstrSet CurrentInstrSet()

    if UsingAArch32\(\) then
        result = if PSTATE.T == '0' then InstrSet\_A32 else InstrSet\_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet\_A64;
    return result;
```

## Library pseudocode for shared/functions/system/CurrentPL

```
// CurrentPL()
// =====

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);
```

## Library pseudocode for shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

## Library pseudocode for shared/functions/system/EL2Enabled

```
// EL2Enabled()
// =====
// Returns TRUE if EL2 is present and executing in either non-Secure state when Secure EL2 is not implemented
// or in Secure state when Secure EL2 is implemented, FALSE otherwise

boolean EL2Enabled()
    return IsSecureEL2Enabled() || (HaveEL(EL2) && !IsSecure());
```

## Library pseudocode for shared/functions/system/ELFromM32

```
// ELFromM32()
// =====

(boolean, bits(2)) ELFromM32(bits(5) mode)
// Convert an AArch32 mode encoding to an Exception level.
// Returns (valid, EL):
// 'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
// and the current value of SCR.NS/SCR_EL3.NS.
// 'EL' is the Exception level decoded from 'mode'.
bits(2) el;
boolean valid = !BadMode(mode); // Check for modes that are not valid for this implementation
case mode of
    when M32_Monitor
        el = EL3;
    when M32_Hyp
        el = EL2;
        valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
    when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
        // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
        // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
        // AArch64, then these modes are EL1 modes.
        el = (if HaveEL(EL3) && HighestELUsingAArch32() && SCR.NS == '0' then EL3 else EL1);
    when M32_User
        el = EL0;
    otherwise
        valid = FALSE; // Passed an illegal mode value
if !valid then el = bits(2) UNKNOWN;
return (valid, el);
```

## Library pseudocode for shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// =====

// Convert an SPSR value encoding to an Exception level.
// Returns (valid, EL):
// 'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
// 'EL' is the Exception level decoded from 'spsr'.

(boolean, bits(2)) ELFromSPSR(bits(32) spsr)
if spsr<4> == '0' then // AArch64 state
    el = spsr<3:2>;
    if HighestELUsingAArch32() then // No AArch64 support
        valid = FALSE;
    elseif !HaveEL(el) then // Exception level not implemented
        valid = FALSE;
    elseif spsr<1> == '1' then // M[1] must be 0
        valid = FALSE;
    elseif el == EL0 && spsr<0> == '1' then // for EL0, M[0] must be 0
        valid = FALSE;
    elseif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
        valid = FALSE; // Unless Secure EL2 is enabled, EL2 only valid in Non-
    else
        valid = TRUE;
elseif !HaveAnyAArch32() then // AArch32 not supported
    valid = FALSE;
else // AArch32 state
    (valid, el) = ELFromM32(spsr<4:0>);
if !valid then el = bits(2) UNKNOWN;
return (valid, el);
```

## Library pseudocode for shared/functions/system/ELIsInHost

```
// ELIsInHost()
// =====

boolean ELIsInHost(bits(2) el)
    return ((IsSecureEL2Enabled\(\) || !IsSecureBelowEL3\(\)) && HaveVirtHostExt\(\) && !ELUsingAArch32\(EL2\)
        HCR_EL2.E2H == '1' && (el == EL2 || (el == EL0 && HCR_EL2.TGE == '1')));
```

## Library pseudocode for shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

## Library pseudocode for shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// =====

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    // Returns (known, aarch32):
    //   'known' is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
    //           using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
    //   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
    boolean aarch32;
    known = TRUE;
    if !HaveAArch32EL(el) then
        aarch32 = FALSE; // Exception level is using AArch64
    elseif HighestELUsingAArch32\(\) then
        aarch32 = TRUE; // All levels are using AArch32
    else
        aarch32_below_el3 = HaveEL\(EL3\) && SCR_EL3.RW == '0';

        aarch32_at_el1 = (aarch32_below_el3 || (HaveEL\(EL2\) &&
            ((HaveSecureEL2Ext\(\) && SCR_EL3.EEL2 == '1') || !secure) && HCR_EL2.RW == '0'
            ! (HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt\(\))));
        if el == EL0 && !aarch32_at_el1 then // Only know if EL0 using AArch32 from PSTATE
            if PSTATE.EL == EL0 then
                aarch32 = PSTATE.nRW == '1'; // EL0 controlled by PSTATE
            else
                known = FALSE; // EL0 state is UNKNOWN
        else
            aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});
        if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);
```

## Library pseudocode for shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// =====

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3\(\));
```

## Library pseudocode for shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =====

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

## Library pseudocode for shared/functions/system/EndOfInstruction

```
// Terminate processing of the current instruction.
EndOfInstruction();
```

## Library pseudocode for shared/functions/system/EnterLowPowerState

```
// PE enters a low-power state
EnterLowPowerState();
```

## Library pseudocode for shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

## Library pseudocode for shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// =====
// Return a PSR value which represents the current PSTATE

bits(32) GetPSRFromPSTATE()
    bits(32) spsr = Zeros();
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HaveDITEExt() then spsr<24> = PSTATE.DIT;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<21> = PSTATE.SS;
    spsr<20> = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        spsr<27> = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if HaveSSBSEExtHaveUAOExt() then spsr<23> = PSTATE.SSBS;
        () then spsr<23> = '0';
        spsr<19:16> = PSTATE.GE;
        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9> = PSTATE.E;
        spsr<8:6> = PSTATE.<A,I,F>; // No PSTATE.D in AArch32 state
        spsr<5> = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW; // bit [4] is the discriminator
        spsr<4:0> = PSTATE.M;
    else // AArch64 state
        if HaveUAOExt() then spsr<23> = PSTATE.UAO;
        if HaveSSBSEExt() then spsr<12> = PSTATE.SSBS;
        if HaveMTEEExt() then spsr<25> = PSTATE.TCO;
        if HaveBTIEExt() then spsr<11:10> = PSTATE.BTYPE;
        () then spsr<23> = PSTATE.UAO;
        spsr<9:6> = PSTATE.<D,A,I,F>;
        spsr<4> = PSTATE.nRW;
        spsr<3:2> = PSTATE.EL;
        spsr<0> = PSTATE.SP;
    return spsr;
```

## Library pseudocode for shared/functions/system/HasArchVersion

```
// HasArchVersion()
// =====
// Return TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// =====

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE; // The Exception level is not implemented
    elseif !HaveAnyAArch32() then
        return FALSE; // No Exception level can use AArch32
    elseif HighestELUsingAArch32() then
        return TRUE; // All Exception levels are using AArch32
    elseif el == HighestEL() then
        return FALSE; // The highest Exception level is using AArch64
    elseif el == EL0 then
        return TRUE; // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/system/HaveAnyAArch32

```
// HaveAnyAArch32()
// =====
// Return TRUE if AArch32 state is supported at any Exception level

boolean HaveAnyAArch32()
    return boolean IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/system/HaveAnyAArch64

```
// HaveAnyAArch64()
// =====
// Return TRUE if AArch64 state is supported at any Exception level

boolean HaveAnyAArch64()
    return !HighestELUsingAArch32();
```

## Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// =====
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    if el IN {EL1, EL0} then
        return TRUE; // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =====
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/system/HighestEL

```
// HighestEL()
// =====
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

## Library pseudocode for shared/functions/system/HighestELUsingAArch32

```
// HighestELUsingAArch32()
// =====
// Return TRUE if configured to boot into AArch32 operation

boolean HighestELUsingAArch32()
    if !HaveAnyAArch32() then return FALSE;
    return boolean IMPLEMENTATION_DEFINED; // e.g. CFG32SIGNAL == HIGH
```

## Library pseudocode for shared/functions/system/Hint\_Yield

```
Hint_Yield();
```

## Library pseudocode for shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// =====

boolean IllegalExceptionReturn(bits(32) spsr)

// Check for illegal return:
// * To an unimplemented Exception level.
// * To EL2 in Secure state, when SecureEL2 is not enabled.
// * To EL0 using AArch64 state, with SPSR.M[0]==1.
// * To AArch64 state with SPSR.M[1]==1.
// * To AArch32 state with an illegal value of SPSR.M.
(valid, target) = ELFromSPSR(spsr);
if !valid then return TRUE;

// Check for return to higher Exception level
if UInt(target) > UInt(PSTATE.EL) then return TRUE;

spsr_mode_is_aarch32 = (spsr<4> == '1');

// Check for illegal return:
// * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
//   Execution state used in the Exception level being returned to, as determined by
//   the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
// * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
//   SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
// * To AArch64 state from AArch32 state (should be caught by above)
(known, target_el_is_aarch32) = ELUsingAArch32K(target);
assert known || (target == EL0 && !ELUsingAArch32(EL1));
if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

// Check for illegal return from AArch32 to AArch64
if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

// Check for illegal return to EL1 when HCR.TGE is set and when either of
// * SecureEL2 is enabled.
// * SecureEL2 is not enabled and EL1 is in Non-secure state.
if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
    if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
return FALSE;
```

## Library pseudocode for shared/functions/system/InstrSet

```
enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

## Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

```
InstructionSynchronizationBarrier();
```

## Library pseudocode for shared/functions/system/InterruptPending

```
// InterruptPending()
// =====
// Return TRUE if there are any pending physical or virtual interrupts, and FALSE otherwise

boolean InterruptPending()
    return IsPhysicalSErrorPending() || IsVirtualSErrorPending();
```

## Library pseudocode for shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// =====
// Return TRUE if the Event Register of this PE is set, and FALSE otherwise

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

### Library pseudocode for shared/functions/system/IsHighestEL

```
// IsHighestEL()
// =====
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;
```

### Library pseudocode for shared/functions/system/IsInHost

```
// IsInHost()
// =====

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

### Library pseudocode for shared/functions/system/IsPhysicalErrorPending

```
// Return TRUE if a physical SError interrupt is pending
boolean IsPhysicalErrorPending();
```

### Library pseudocode for shared/functions/system/IsSecure

```
// IsSecure()
// =====

boolean IsSecure()
    // Return TRUE if current Exception level is in Secure state.
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elseif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32\_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
```

### Library pseudocode for shared/functions/system/IsSecureBelowEL3

```
// IsSecureBelowEL3()
// =====

// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL(EL3) then
        return SCR\_GEN[][.NS == '0'];
    elseif HaveEL(EL2) && (!HaveSecureEL2Ext() || HighestELUsingAArch32()) then
        // If Secure EL2 is not an architecture option then we must be Non-secure.
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```



## Library pseudocode for shared/functions/system/IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// =====
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise

boolean IsSecureEL2Enabled()
    return ( HaveSecureEL2Ext() && HaveEL(EL2) && !ELUsingAArch32(EL2) &&
        ((HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1') ||
        (!HaveEL(EL3) && IsSecure())) );
```

## Library pseudocode for shared/functions/system/IsVirtualSErrorPending

```
// Return TRUE if a virtual SError interrupt is pending
boolean IsVirtualSErrorPending();
```

## Library pseudocode for shared/functions/system/Mode\_Bits

```
constant bits(5) M32_User      = '10000';
constant bits(5) M32_FIQ      = '10001';
constant bits(5) M32_IRQ      = '10010';
constant bits(5) M32_Svc      = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort    = '10111';
constant bits(5) M32_Hyp      = '11010';
constant bits(5) M32_Undef    = '11011';
constant bits(5) M32_System   = '11111';
```

## Library pseudocode for shared/functions/system/PLOfEL

```
// PLOfEL()
// =====

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3    return if HighestELUsingAArch32() then PL1 else PL3;
        when EL2    return PL2;
        when EL1    return PL1;
        when EL0    return PL0;
```

## Library pseudocode for shared/functions/system/PSTATE

```
ProcState PSTATE;
```

## Library pseudocode for shared/functions/system/PrivilegeLevel

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

## Library pseudocode for shared/functions/system/ProcState

```
type ProcState is (  
    bits (1) N,          // Negative condition flag  
    bits (1) Z,          // Zero condition flag  
    bits (1) C,          // Carry condition flag  
    bits (1) V,          // oVerflow condition flag  
    bits (1) D,          // Debug mask bit [AArch64 only]  
    bits (1) A,          // SError interrupt mask bit  
    bits (1) I,          // IRQ mask bit  
    bits (1) F,          // FIQ mask bit  
    bits (1) PAN,        // Privileged Access Never Bit [v8.1]  
    bits (1) UAO,        // User Access Override [v8.2]  
    bits (1) DIT,        // Data Independent Timing [v8.4]  
    bits (1) TCO,        // Tag Check Override [v8.5, AArch64 only]  
    bits (2) BTYPE,      // Branch Type [v8.5]  
    bits (1) SS,         // Software step bit  
    bits (1) IL,         // Illegal Execution state bit  
    bits (2) EL,         // Exception Level  
    bits (1) nRW,        // not Register Width: 0=64, 1=32  
    bits (1) SP,         // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]  
    bits (1) Q,          // Cumulative saturation flag [AArch32 only]  
    bits (4) GE,         // Greater than or Equal flags [AArch32 only]  
    bits (1) SSBS,       // Speculative Store Bypass Safe  
    bits (8) IT,         // If-then bits, RES0 in CPSR [AArch32 only]  
    bits (1) J,          // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]  
    bits (1) T,          // T32 bit, RES0 in CPSR [AArch32 only]  
    bits (1) E,          // Endianness bit [AArch32 only]  
    bits (5) M           // Mode field [AArch32 only]  
)
```

## Library pseudocode for shared/functions/system/RandomTag

```
// RandomTag()  
// =====  
// Generate a random Allocation Tag.  
  
bits(4) RandomTag()  
    bits(4) tag;  
    for i = 0 to 3  
        tag<i> = NextRandomTagBit();  
    return tag;
```

## Library pseudocode for shared/functions/system/RandomTagBit

```
// RandomTagBit()  
// =====  
// Generate a random bit suitable for generating a random Allocation Tag.  
  
bit NextRandomTagBit()  
    bits(16) lfsr = RGSr_EL1.SEED;  
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;  
    RGSr_EL1.SEED = top:lfsr<15:1>;  
    return top;
```

## Library pseudocode for shared/functions/system/RestoredITBits

```
// RestoredITBits()
// =====
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(32) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable\_ILZEROIT) then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

## Library pseudocode for shared/functions/system/SCRType

```
type SCRType;
```

## Library pseudocode for shared/functions/system/SCR\_GEN

```
// SCR_GEN[]
// =====

SCRType SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(32) r;
    if HighestELUsingAArch32() then
        r = SCR;
    else
        r = SCR_EL3;
    return r;
```

## Library pseudocode for shared/functions/system/SendEvent

```
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed
SendEvent();
```

## Library pseudocode for shared/functions/system/SendEventLocal

```
// SendEventLocal()
// =====
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed

SendEventLocal()
    EventRegister = '1';
    return;
```

## Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// =====
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(32) spsr)
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then
            AArch32.WriteMode(spsr<4:0>); // AArch32 state
            // Sets PSTATE.EL correctly
        else
            // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL = spsr<3:2>;
            PSTATE.SP = spsr<0>;
        // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
        // the T bit is set to zero or copied from SPSR.
        if PSTATE.IL == '1' && PSTATE.nRW == '1' then
            if ConstrainUnpredictableBool(Unpredictable\_ILZEROT) then spsr<5> = '0';

        // State that is reinstated regardless of illegal exception return
        PSTATE.<N,Z,C,V> = spsr<31:28>;
        if if PSTATE.nRW == '1' then // AArch32 state
            PSTATE.Q = spsr<27>;
            PSTATE.IT = HaveDITEExt() then PSTATE.DIT = spsr<24>;
        if PSTATE.nRW == '1' then // AArch32 state
            PSTATE.Q = spsr<27>;
            PSTATE.IT = RestoredITBits(spsr);
            ShouldAdvanceIT = FALSE;
            PSTATE.GE = spsr<19:16>;
            PSTATE.E = spsr<9>;
            PSTATE.<A,I,F> = spsr<8:6>; // No PSTATE.D in AArch32 state
            PSTATE.T = spsr<5>; // PSTATE.J is RES0
        else // AArch64 state
            PSTATE.<D,A,I,F> = spsr<9:6>; // No PSTATE.<Q,IT,GE,E,T> in AArch64 state
            if HaveSSBSEExt() then PSTATE.SSBS = spsr<23>;
            PSTATE.GE = spsr<19:16>;
            PSTATE.E = spsr<9>;
            PSTATE.<A,I,F> = spsr<8:6>; // No PSTATE.D in AArch32 state
            PSTATE.T = spsr<5>; // PSTATE.J is RES0
        else // AArch64 state
            if HaveUAOExt() then PSTATE.UAO = spsr<23>;
            if HaveSSBSEExt() then PSTATE.SSBS = spsr<12>;
            PSTATE.<D,A,I,F> = spsr<9:6>;

            if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;

            if HavePANExt() then PSTATE.PAN = spsr<22>;
            if HaveMTEEExt() then
                if PSTATE.nRW != '1' then
                    PSTATE.TCO = spsr<25>;
            () then PSTATE.<PAN> = spsr<22>;
    return;
```

## Library pseudocode for shared/functions/system/ShouldAdvanceIT

```
boolean ShouldAdvanceIT;
```

## Library pseudocode for shared/functions/system/SpeculationBarrier

```
SpeculationBarrier();
```

### Library pseudocode for shared/functions/system/SynchronizeContext

```
SynchronizeContext();
```

### Library pseudocode for shared/functions/system/SynchronizeErrors

```
// Implements the error synchronization event.  
SynchronizeErrors();
```

### Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt  
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

### Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt or unmasked virtual SError  
// interrupt.  
TakeUnmaskedSErrorInterrupts();
```

### Library pseudocode for shared/functions/system/ThisInstr

```
bits(32) ThisInstr();
```

### Library pseudocode for shared/functions/system/ThisInstrLength

```
integer ThisInstrLength();
```

### Library pseudocode for shared/functions/system/Unreachable

```
Unreachable()  
    assert FALSE;
```

### Library pseudocode for shared/functions/system/UsingAArch32

```
// UsingAArch32()  
// =====  
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.  
  
boolean UsingAArch32()  
    boolean aarch32 = (PSTATE.nRW == '1');  
    if !HaveAnyAArch32() then assert !aarch32;  
    if HighestELUsingAArch32() then assert aarch32;  
    return aarch32;
```

### Library pseudocode for shared/functions/system/WaitForEvent

```
// WaitForEvent()  
// =====  
// PE suspends its operation and enters a low-power state  
// if the Event Register is clear when the WFE is executed  
  
WaitForEvent()  
    if EventRegister == '0' then  
        EnterLowPowerState();  
    return;
```

## Library pseudocode for shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()  
// =====  
// PE suspends its operation to enter a low-power state  
// until a WFI wake-up event occurs or the PE is reset  
  
WaitForInterrupt()  
    EnterLowPowerState();  
    return;
```



```

// ConstrainUnpredictable()
// =====
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.

Constraint ConstrainUnpredictable(Unpredictable which)
    case which of
        when Unpredictable WBOVERLAPLD
            return Constraint WBSUPPRESS; // return loaded value
        when Unpredictable WBOVERLAPST
            return Constraint NONE; // store pre-writeback value
        when Unpredictable LDPOVERLAP
            return Constraint UNDEF; // instruction is UNDEFINED
        when Unpredictable BASEOVERLAP
            return Constraint NONE; // use original address
        when Unpredictable DATAOVERLAP
            return Constraint NONE; // store original value
        when Unpredictable DEVPAGE2
            return Constraint FAULT; // take an alignment fault
        when Unpredictable INSTRDEVICE
            return Constraint NONE; // Do not take a fault
        when Unpredictable RESCPACR
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESMAIR
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESTEXCB
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESDACR
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESPRRR
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESVTCRS
            return Constraint UNKNOWN; // Map to UNKNOWN value
        when Unpredictable RESTnSZ
            return Constraint FORCE; // Map to the limit value
        when Unpredictable OORTnSZ
            return Constraint FORCE; // Map to the limit value
        when Unpredictable LARGEIPA
            return Constraint FORCE; // Restrict the inputs size to the PAMax value
        when Unpredictable ESRCONDPASS
            return Constraint FALSE; // Report as "AL"
        when Unpredictable ILZEROIT
            return Constraint FALSE; // Do not zero PSTATE.IT
        when Unpredictable ILZEROT
            return Constraint FALSE; // Do not zero PSTATE.T
        when Unpredictable BPVECTORCATCHPRI
            return Constraint TRUE; // Debug Vector Catch: match on 2nd halfword
        when Unpredictable VCMATCHHALF
            return Constraint FALSE; // No match
        when Unpredictable VCMATCHDAPA
            return Constraint FALSE; // No match on Data Abort or Prefetch abort
        when Unpredictable WPMASKANDBAS
            return Constraint FALSE; // Watchpoint disabled
        when Unpredictable WPBASCONTIGUOUS
            return Constraint FALSE; // Watchpoint disabled
        when Unpredictable RESWPMASK
            return Constraint DISABLED; // Watchpoint disabled
        when Unpredictable WPMASKEDBITS
            return Constraint FALSE; // Watchpoint disabled
        when Unpredictable RESBPWPCTRL
            return Constraint DISABLED; // Breakpoint/watchpoint disabled
        when Unpredictable BPNOTIMPL
            return Constraint DISABLED; // Breakpoint disabled

```



```

when Unpredictable RESBPTYPE
    return Constraint DISABLED; // Breakpoint disabled
when Unpredictable BPNOTCTXCMP
    return Constraint DISABLED; // Breakpoint disabled
when Unpredictable BPMATCHHALF
    return Constraint FALSE; // No match
when Unpredictable BPMISMATCHHALF
    return Constraint FALSE; // No match
when Unpredictable RESTARTALIGNPC
    return Constraint FALSE; // Do not force alignment
when Unpredictable RESTARTZEROUPPERPC
    return Constraint TRUE; // Force zero extension
when Unpredictable ZEROUPPER
    return Constraint TRUE; // zero top halves of X registers
when Unpredictable ERETZEROUPPERPC
    return Constraint TRUE; // zero top half of PC
when Unpredictable A32FORCEALIGNPC
    return Constraint FALSE; // Do not force alignment
when Unpredictable SMD
    return Constraint UNDEF; // disabled SMC is Unallocated
when Unpredictable NONFAULT
    return Constraint FALSE; // Speculation enabled
when Unpredictable SVEZEROUPPER
    return Constraint TRUE; // zero top bits of Z registers
when Unpredictable SVELDNFDATA
    return Constraint TRUE; // Load mem data in NF loads
when Unpredictable SVELDNFZERO
    return Constraint TRUE; // Write zeros in NF loads
when Unpredictable AFUPDATE
    return Constraint TRUE; // AF update for alignment or permission fault
when Unpredictable IESBinDebug // Use SCTL[[]].IESB in Debug state
    return Constraint TRUE;
when Unpredictable ZEROBTYP
    return Constraint TRUE; // Save BTYPE in SPSR_ELx/DPSR_EL0 as '00'
when Unpredictable CLEARERRITEZERO // Clearing sticky errors when instruction in flight
    return Constraint FALSE;

```

## Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

```

// ConstrainUnpredictableBits()
// =====

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint, bits(width)) ConstrainUnpredictableBits(Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint_UNKNOWN then
        return (c, Zeros(width)); // See notes; this is an example implementation only
    else
        return (c, bits(width) UNKNOWN); // bits result not used

```

## Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// =====

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

boolean ConstrainUnpredictableBool(Unpredictable which)

    c = ConstrainUnpredictable(which);
    assert c IN {Constraint\_TRUE, Constraint\_FALSE};
    return (c == Constraint\_TRUE);
```

## Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// =====

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the integer part
// of the result.

(Constraint, integer) ConstrainUnpredictableInteger(integer low, integer high, Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint\_UNKNOWN then
        return (c, low); // See notes; this is an example implementation only
    else
        return (c, integer UNKNOWN); // integer result not used
```

## Library pseudocode for shared/functions/unpredictable/Constraint

```
enumeration Constraint    { // General
    Constraint_NONE,        // Instruction executes with
                           // no change or side-effect to its described
    Constraint_UNKNOWN,    // Destination register has UNKNOWN value
    Constraint_UNDEF,      // Instruction is UNDEFINED
    Constraint_UNDEFEL0,   // Instruction is UNDEFINED at EL0 only
    Constraint_NOP,        // Instruction executes as NOP
    Constraint_TRUE,
    Constraint_FALSE,
    Constraint_DISABLED,
    Constraint_UNCOND,     // Instruction executes unconditionally
    Constraint_COND,       // Instruction executes conditionally
    Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
    // Load-store
    Constraint_WBSUPPRESS, Constraint_FAULT,
    // IPA too large
    Constraint_FORCE, Constraint_FORCENOSLCHECK};
```



```

enumeration Unpredictable { // Writeback/transfer register overlap (load)
    Unpredictable_WBOVERLAPLD,
    // Writeback/transfer register overlap (store)
    Unpredictable_WBOVERLAPST,
    // Load Pair transfer register overlap
    Unpredictable_LDPOVERLAP,
    // Store-exclusive base/status register overlap
    Unpredictable_BASEOVERLAP,
    // Store-exclusive data/status register overlap
    Unpredictable_DATAOVERLAP,
    // Load-store alignment checks
    Unpredictable_DEVPAGE2,
    // Instruction fetch from Device memory
    Unpredictable_INSTRDEVICE,
    // Reserved CPACR value
    Unpredictable_RESCPACR,
    // Reserved MAIR value
    Unpredictable_RESMAIR,
    // Reserved TEX:C:B value
    Unpredictable_RESTEXCB,
    // Reserved PRRR value
    Unpredictable_RESPPRR,
    // Reserved DACR field
    Unpredictable_RESDACR,
    // Reserved VTCR.S value
    Unpredictable_RESVTCRS,
    // Reserved TCR.TnSZ value
    Unpredictable_RESTnSZ,
    // Out-of-range TCR.TnSZ value
    Unpredictable_OORTnSZ,
    // IPA size exceeds PA size
    Unpredictable_LARGEIPA,
    // Syndrome for a known-passing conditional A32 instruction
    Unpredictable_ESRCONDPASS,
    // Illegal State exception: zero PSTATE.IT
    Unpredictable_ILZEROIT,
    // Illegal State exception: zero PSTATE.T
    Unpredictable_ILZEROT,
    // Debug: prioritization of Vector Catch
    Unpredictable_BPVECTORCATCHPRI,
    // Debug Vector Catch: match on 2nd halfword
    Unpredictable_VCMATCHHALF,
    // Debug Vector Catch: match on Data Abort or Prefetch abort
    Unpredictable_VCMATCHDAPA,
    // Debug watchpoints: non-zero MASK and non-ones BAS
    Unpredictable_WPMASKANDBAS,
    // Debug watchpoints: non-contiguous BAS
    Unpredictable_WPBASCONTIGUOUS,
    // Debug watchpoints: reserved MASK
    Unpredictable_RESWPMASK,
    // Debug watchpoints: non-zero MASKed bits of address
    Unpredictable_WPMASKEDBITS,
    // Debug breakpoints and watchpoints: reserved control bits
    Unpredictable_RESBPWPCTRL,
    // Debug breakpoints: not implemented
    Unpredictable_BPNOTIMPL,
    // Debug breakpoints: reserved type
    Unpredictable_RESBPTYPE,
    // Debug breakpoints: not-context-aware breakpoint
    Unpredictable_BPNOTCTXCMP,
    // Debug breakpoints: match on 2nd halfword of instruction
    Unpredictable_BPMATCHHALF,
    // Debug breakpoints: mismatch on 2nd halfword of instruction
    Unpredictable_BPMISMATCHHALF,
    // Debug: restart to a misaligned AArch32 PC value
    Unpredictable_RESTARTALIGNPC,
    // Debug: restart to a not-zero-extended AArch32 PC value
    Unpredictable_RESTARTZEROUPPERPC,
    // Zero top 32 bits of X registers in AArch32 state
    Unpredictable_ZEROUPPER,

```

```

// Zero top 32 bits of PC on illegal return to AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// FF speculation
Unpredictable_NONFAULT,
// Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,
// Load mem data in NF loads
Unpredictable_SVELDNFDATA,
// Write zeros in NF loads
Unpredictable_SVELDNFZERO,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Consider SCTL[ ].IESB in Debug state
Unpredictable_IESBinDebug,
// No events selected in PMSEVFR_EL1
Unpredictable_ZEROPMSEVFR,
// No operation type selected in PMSFCR_EL1
Unpredictable_NOOPTYPES,
// Zero latency in PMSLATFR_EL1
Unpredictable_ZEROMINLATENCY,
// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO};

```

## Library pseudocode for shared/functions/vector/AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// =====

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
case cmode<3:1> of
  when '000'
    imm64 = Replicate(Zeros(24):imm8, 2);
  when '001'
    imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
  when '010'
    imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
  when '011'
    imm64 = Replicate(imm8:Zeros(24), 2);
  when '100'
    imm64 = Replicate(Zeros(8):imm8, 4);
  when '101'
    imm64 = Replicate(imm8:Zeros(8), 4);
  when '110'
    if cmode<0> == '0' then
      imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
    else
      imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
  when '111'
    if cmode<0> == '0' && op == '0' then
      imm64 = Replicate(imm8, 8);
    if cmode<0> == '0' && op == '1' then
      imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
      imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
      imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
      imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
      imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
    if cmode<0> == '1' && op == '0' then
      imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 5):imm8<5:0>:Zeros(19);
      imm64 = Replicate(imm32, 2);
    if cmode<0> == '1' && op == '1' then
      if UsingAArch32() then ReservedEncoding();
      imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 8):imm8<5:0>:Zeros(48);

return imm64;
```

## Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// =====

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
result = Zeros(M+N);
extended_op2 = ZeroExtend(op2, M+N);
for i=0 to M-1
  if op1<i> == '1' then
    result = result EOR LSL(extended_op2, i);
return result;
```

## Library pseudocode for shared/functions/vector/SatQ

```
// SatQ()
// =====

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
(result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
return (result, sat);
```

## Library pseudocode for shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// =====

(bits(N), boolean) SignedSatQ(integer i, integer N)
  if i > 2^(N-1) - 1 then
    result = 2^(N-1) - 1; saturated = TRUE;
  elsif i < -(2^(N-1)) then
    result = -(2^(N-1)); saturated = TRUE;
  else
    result = i; saturated = FALSE;
  return (result<N-1:0>, saturated);
```

## Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =====

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
  assert N IN {16,32};
  if operand<N-1:N-2> == '00' then // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
    result = Ones(N);
  else
    // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)

    // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
    case N of
      when 16 estimate = RecipSqrtEstimate(UInt(operand<15:7>));
      when 32 estimate = RecipSqrtEstimate(UInt(operand<31:23>));

    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate<8:0> : Zeros(N-9);

  return result;
```

## Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =====

bits(N) UnsignedRecipEstimate(bits(N) operand)
  assert N IN {16,32};
  if operand<N-1> == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
    result = Ones(N);
  else
    // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

    // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
    case N of
      when 16 estimate = RecipEstimate(UInt(operand<15:7>));
      when 32 estimate = RecipEstimate(UInt(operand<31:23>));

    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate<8:0> : Zeros(N-9);

  return result;
```

## Library pseudocode for shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// =====

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
  if i > 2^N - 1 then
    result = 2^N - 1; saturated = TRUE;
  elsif i < 0 then
    result = 0; saturated = TRUE;
  else
    result = i; saturated = FALSE;
  return (result<N-1:0>, saturated);
```

## Library pseudocode for shared/translation/attrs/CombineS1S2AttrHints

```
// CombineS1S2AttrHints()
// =====

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc)

  MemAttrHints result;

  if if s2desc.attrs == '01' || s1desc.attrs == '01' then
    result.attrs = bits(2) UNKNOWN; // Reserved
  elsif s2desc.attrs == HaveStage2MemAttrControl() && HCR_EL2.FWB == '1' then
    if s2desc.attrs == MemAttr WB then
      result.attrs = s1desc.attrs;
    elsif s2desc.attrs == MemAttr WT then
      result.attrs = MemAttr WB;
    else
      result.attrs = MemAttr NC;
  else
    if s2desc.attrs == '01' || s1desc.attrs == '01' then
      result.attrs = bits(2) UNKNOWN; // Reserved
    elsif s2desc.attrs == MemAttr NC || s1desc.attrs == MemAttr NC then
      result.attrs = MemAttr NC; // Non-cacheable
    elsif s2desc.attrs == MemAttr WT || s1desc.attrs == MemAttr WT then
      result.attrs = MemAttr WT; // Write-through
    else
      result.attrs = MemAttr WB; // Write-back

  result.hints = s1desc.hints;
  result.transient = s1desc.transient;

  return result;
```



## Library pseudocode for shared/translation/attrs/CombineS1S2Desc

```
// CombineS1S2Desc()
// =====
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    AddressDescriptor result;

    result.paddress = s2desc.paddress;

    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    elsif s2desc.memattrs.type == MemType_Device || s1desc.memattrs.type == MemType_Device then
        result.memattrs.type = MemType_Device;
        if s1desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s2desc.memattrs.device;
        elsif s2desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s1desc.memattrs.device;
        else
            // Both Device
            result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
                                                    s2desc.memattrs.device);
    result.memattrs.tagged = FALSE;
    else
        // Both Normal
        result.memattrs.type = MemType_Normal;
        result.memattrs.device = DeviceType_UNKNOWN;
        result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner);
        result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer);
        result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
        result.memattrs.outershareable = (s1desc.memattrs.outershareable ||
                                           s2desc.memattrs.outershareable);
    result.memattrs.tagged = (s1desc.memattrs.tagged &&
                             result.memattrs.inner.attrs ==
                             result.memattrs = MemAttr_WB &&
                             result.memattrs.inner.hints == MemHint_RWA &&
                             result.memattrs.outer.attrs == MemAttr_WB &&
                             result.memattrs.outer.hints == MemHint_RWA);

    result.memattrs = MemAttrDefaults(result.memattrs);

    return result;
```

## Library pseudocode for shared/translation/attrs/CombineS1S2Device

```
// CombineS1S2Device()
// =====
// Combines device types from stage 1 and stage 2

DeviceType CombineS1S2Device(DeviceType s1device, DeviceType s2device)

    if s2device == DeviceType_nGnRnE || s1device == DeviceType_nGnRnE then
        result = DeviceType_nGnRnE;
    elsif s2device == DeviceType_nGnRE || s1device == DeviceType_nGnRE then
        result = DeviceType_nGnRE;
    elsif s2device == DeviceType_nGRE || s1device == DeviceType_nGRE then
        result = DeviceType_nGRE;
    else
        result = DeviceType_GRE;

    return result;
```

## Library pseudocode for shared/translation/attrs/LongConvertAttrsHints

```
// LongConvertAttrsHints()
// =====
// Convert the long attribute fields for Normal memory as used in the MAIR fields
// to orthogonal attributes and hints

MemAttrHints LongConvertAttrsHints(bits(4) attrfield, AccType acctype)
    assert !IsZero(attrfield);
    MemAttrHints result;
    if S1CacheDisabled(acctype) then                // Force Non-cacheable
        result.attrs = MemAttr\_NC;
        result.hints = MemHint\_No;
    else
        if attrfield<3:2> == '00' then                // Write-through transient
            result.attrs = MemAttr\_WT;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        elsif attrfield<3:0> == '0100' then          // Non-cacheable (no allocate)
            result.attrs = MemAttr\_NC;
            result.hints = MemHint\_No;
            result.transient = FALSE;
        elsif attrfield<3:2> == '01' then            // Write-back transient
            result.attrs = MemAttr\_WB;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        else                                          // Write-through/Write-back non-transient
            result.attrs = attrfield<3:2>;
            result.hints = attrfield<1:0>;
            result.transient = FALSE;

    return result;
```

## Library pseudocode for shared/translation/attrs/MemAttrDefaults

```
// MemAttrDefaults()
// =====
// Supply default values for memory attributes, including overriding the shareability attributes
// for Device and Non-cacheable memory types.

MemoryAttributes MemAttrDefaults(MemoryAttributes memattrs)

    if memattrs.type == MemType\_Device then
        memattrs.inner = MemAttrHints UNKNOWN;
        memattrs.outer = MemAttrHints UNKNOWN;
        memattrs.shareable = TRUE;
        memattrs.outershareable = TRUE;
    else
        memattrs.device = DeviceType UNKNOWN;
        if memattrs.inner.attrs == MemAttr\_NC && memattrs.outer.attrs == MemAttr\_NC then
            memattrs.shareable = TRUE;
            memattrs.outershareable = TRUE;

    return memattrs;
```

## Library pseudocode for shared/translation/attrs/S1CacheDisabled

```
// S1CacheDisabled()
// =====

boolean S1CacheDisabled(AccType acctype)
    if ELUsingAArch32(S1TranslationRegime()) then
        if PSTATE.EL == EL2 then
            enable = if acctype == AccType_IFETCH then HSCTLR.I else HSCTLR.C;
        else
            enable = if acctype == AccType_IFETCH then SCTLR.I else SCTLR.C;
    else
        enable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;
    return enable == '0';
```

## Library pseudocode for shared/translation/attrs/S2AttrDecode

```
// S2AttrDecode()
// =====
// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)

    MemoryAttributes memattrs;

    apply_force_writeback = if attr<3:2> == '00' then // Device
    memattrs.type = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';

    // Device memory
    if (apply_force_writeback && attr<2> == '0') || attr<3:2> == '00' then
        memattrs.type = MemType_Device;
        case attr<1:0> of
            when '00' memattrs.device = DeviceType_nGnRnE;
            when '01' memattrs.device = DeviceType_nGnRE;
            when '10' memattrs.device = DeviceType_nGRE;
            when '11' memattrs.device = DeviceType_GRE;

    // Normal memory
    elseif attr<1:0> != '00' then
    elseif attr<1:0> != '00' then // Normal
        memattrs.type = MemType_Normal;
        if apply_force_writeback then
            memattrs.outer = S2ConvertAttrHints(attr<1:0>, acctype);
        else
            memattrs.outer = ;
        memattrs.outer = S2ConvertAttrHints(attr<3:2>, acctype);
        memattrs.inner = S2ConvertAttrHints(attr<1:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';

    else
        memattrs = MemoryAttributes UNKNOWN; // Reserved

    return MemAttrDefaults(memattrs);
```

## Library pseudocode for shared/translation/attrs/S2CacheDisabled

```
// S2CacheDisabled()
// =====

boolean S2CacheDisabled(AccType acctype)
    if ELUsingAArch32(EL2) then
        disable = if acctype == AccType_IFETCH then HCR2.ID else HCR2.CD;
    else
        disable = if acctype == AccType_IFETCH then HCR_EL2.ID else HCR_EL2.CD;
    return disable == '1';
```

## Library pseudocode for shared/translation/attrs/S2ConvertAttrsHints

```
// S2ConvertAttrsHints()
// =====
// Converts the attribute fields for Normal memory as used in stage 2
// descriptors to orthogonal attributes and hints

MemAttrHints S2ConvertAttrsHints(bits(2) attr, AccType acctype)
    assert !IsZero(attr);

    MemAttrHints result;

    if S2CacheDisabled(acctype) then                // Force Non-cacheable
        result.attrs = MemAttr\_NC;
        result.hints = MemHint\_No;
    else
        case attr of
            when '01'                                // Non-cacheable (no allocate)
                result.attrs = MemAttr\_NC;
                result.hints = MemHint\_No;
            when '10'                                // Write-through
                result.attrs = MemAttr\_WT;
                result.hints = MemHint\_RWA;
            when '11'                                // Write-back
                result.attrs = MemAttr\_WB;
                result.hints = MemHint\_RWA;

    result.transient = FALSE;

    return result;
```

## Library pseudocode for shared/translation/attrs/ShortConvertAttrsHints

```
// ShortConvertAttrsHints()
// =====
// Converts the short attribute fields for Normal memory as used in the TTBR and
// TEX fields to orthogonal attributes and hints

MemAttrHints ShortConvertAttrsHints(bits(2) RGN, AccType acctype, boolean secondstage)

    MemAttrHints result;

    if (!secondstage && S1CacheDisabled(acctype)) || (secondstage && S2CacheDisabled(acctype)) then
        // Force Non-cacheable
        result.attrs = MemAttr\_NC;
        result.hints = MemHint\_No;
    else
        case RGN of
            when '00'                                // Non-cacheable (no allocate)
                result.attrs = MemAttr\_NC;
                result.hints = MemHint\_No;
            when '01'                                // Write-back, Read and Write allocate
                result.attrs = MemAttr\_WB;
                result.hints = MemHint\_RWA;
            when '10'                                // Write-through, Read allocate
                result.attrs = MemAttr\_WT;
                result.hints = MemHint\_RA;
            when '11'                                // Write-back, Read allocate
                result.attrs = MemAttr\_WB;
                result.hints = MemHint\_RA;

    result.transient = FALSE;

    return result;
```

## Library pseudocode for shared/translation/attrs/WalkAttrDecode

```
// WalkAttrDecode()
// =====

MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN, boolean secondstage)

    MemoryAttributes memattrs;

    AccType acctype = AccType\_NORMAL;

    memattrs.type = MemType\_Normal;
    memattrs.inner = ShortConvertAttrsHints(IRGN, acctype, secondstage);
    memattrs.outer = ShortConvertAttrsHints(ORGN, acctype, secondstage);
    memattrs.shareable = SH<1> == '1';
    memattrs.outershareable = SH == '10';
    memattrs.tagged = FALSE;

    return MemAttrDefaults(memattrs);
```

## Library pseudocode for shared/translation/translation/HasS2Translation

```
// HasS2Translation()
// =====
// Returns TRUE if stage 2 translation is present for the current translation regime

boolean HasS2Translation()
    return (EL2Enabled() && !IsInHost() && PSTATE.EL IN {EL0, EL1});
```

## Library pseudocode for shared/translation/translation/Have16bitVMID

```
// Have16bitVMID()
// =====
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return HaveEL(EL2) && boolean IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/translation/translation/PAMax

```
// PAMax()
// =====
// Returns the IMPLEMENTATION_DEFINED upper limit on the physical address
// size for this processor, as log2().

integer PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

## Library pseudocode for shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt\(\) && ELIsInHost\(el\) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

## Library pseudocode for shared/translation/translation/VAMax

```
// VAMax()
// =====
// Returns the IMPLEMENTATION_DEFINED upper limit on the virtual address
// size for this processor, as log2().

integer VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
```

Internal version only: isa [v30.25](#)~~v29.05~~, AdvSIMD [v27.01](#)~~v26.0~~, pseudocode [v85-xml-00](#)~~bet8~~ [rc3](#)~~v35.3~~; Build timestamp: [2018-09-13T13:20:04](#)~~2018-06-16T09:0445~~

Copyright © 2010-2018 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)