

(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.

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 ™ 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 © 2020 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 version 21.0)

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14.5341

Copyright © 2010-2020 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](#), [CASAHL](#), [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 Carry Flag.

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 Sign bits.

[CLZ](#): Count Leading Zeros.

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.

[DGH](#): Data Gathering Hint.

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.

[LD64B](#): Single-copy Atomic 64-byte Load.

[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.

[LDGM](#): Load Tag Multiple.

[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](#), [LDRAB](#): 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).

[PRFUM](#): 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.

[ST64B](#): Single-copy Atomic 64-byte Store without Return.

[ST64BV](#): Single-copy Atomic 64-byte Store with Return.



[ST64BV0](#): Single-copy Atomic 64-byte EL0 Store with Return.

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, LDCLRAH, LDCLRALH, 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.

[STGM](#): Store Tag Multiple.

[STGP](#): Store Allocation Tag and Pair of registers.

[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.

[STZGM](#): Store Tag and Zero Multiple.

[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.

TCANCEL: Cancel current transaction.

TCOMMIT: Commit current transaction.

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).

TSTART: Start transaction.

TTEST: Test transaction state.

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

[UBFM](#): Unsigned Bitfield Move.

UBFX: Unsigned Bitfield Extract: an alias of UBFM.

UDF: 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.

[WFET](#): Wait For Event with Timeout.

[WFI](#): Wait For Interrupt.

[WFIT](#): Wait For Interrupt with Timeout.

XAFLAG: Convert floating-point condition flags from external format to Arm format.

[XPACD](#), [XPACL](#), [XPACLR](#): Strip Pointer Authentication Code.

[YIELD](#): YIELD.

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ADC

Add with Carry adds two register values and the Carry flag 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	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
op S																															

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

ADC <Wd>, <Wn>, <Wm>

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

ADC <Xd>, <Xn>, <Xm>

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

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcvc;

(result, -) =if sub_op then
    operand2 = NOT(operand2);

(result, nzcvc) = AddWithCarry(operand1, operand2, PSTATE.C); {operand1, operand2, PSTATE.C};

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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ADCS

Add with Carry, setting flags, adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.

31	30	29	28	27	26	25	24	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	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
op S																															

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

ADCS <Wd>, <Wn>, <Wm>

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

ADCS <Xd>, <Xn>, <Xm>

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

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcvc;

if sub_op then
    operand2 = NOT(operand2);

(result, nzcvc) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcvc; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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; 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;
```

## 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;

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

(result, nzcv) = AddWithCarry(operand1, operand2, '0');
(operand1, operand2, carry_in);

if d == 31 then 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 \(to/from SP\)](#).

31	30	29	28	27	26	25	24	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	0	1	0	sh	imm12												Rn						Rd			
op S																															

### 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 sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:(imm12-Zeros(12)), datasize);
```

## 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 "sh":

sh	<shift>
0	LSL #0
1	LSL #12

## Alias Conditions

Alias	Is preferred when
<a href="#">MOV (to/from SP)</a>	sh == '0' && 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;

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

(result, nzcv) = AddWithCarry(operand1, imm, '0');
(operand1, operand2, carry_in);

if d == 31 then 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14Z

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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	shift	0	Rm						imm6						Rn						Rd			
op S																															

### 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;

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;

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

(result, nzcv) = AddWithCarry(operand1, operand2, '0'); (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

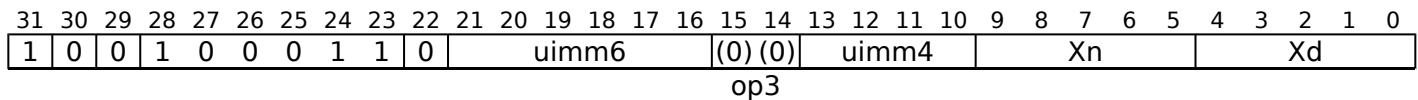
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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 (FEAT\_MTE)



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

```
if !HaveMTEExt() then UNDEFINED;
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 = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
(start_tag, tag_offset, exclude);
else
    rtag = '0000';

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

result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, rtag);

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



(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; 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;
```

## 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, '0');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcvc; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

**(old)**

**htmldiff from-**

**(new)**

## 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 \(immediate\)](#).

31	30	29	28	27	26	25	24	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	0	1	0	sh		imm12										Rn						Rd			
op S																															

### 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 sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:(imm12-Zeros(12)), datasize);
```

## 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 "sh":

sh	<shift>
0	LSL #0
1	LSL #12

## 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, imm, '0');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcv;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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 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 \(shifted register\)](#).

31	30	29	28	27	26	25	24	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	shift	0	Rm						imm6						Rn						Rd			
op S																															

### 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;

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
CMN (shifted register)	Rd == '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, '0');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcv; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

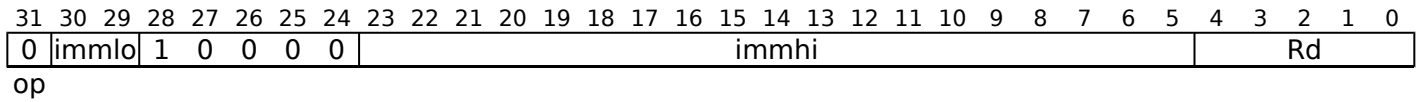
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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



## ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.



ADR <Xd>, <label>

```
integer d = UInt(Rd);
boolean page = (op == '1');
bits(64) imm;

imm = if page then
    imm = SignExtend(immhi:immlo, 64); (immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(immhi:immlo, 64);
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <label> Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in "immhi:immlo".

## Operation

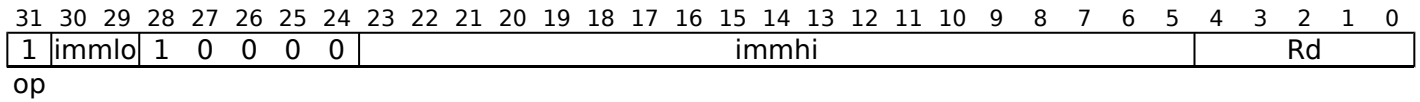
```
bits(64) base = PC[1];
if page then
    base<11:0> =
    Zeros(12);
X[d] = base + imm;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ADRP

Form PC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.



ADRP <Xd>, <label>

```
integer d = UInt(Rd);
boolean page = (op == '1');
bits(64) imm;

imm = if page then
    imm = SignExtend(immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(immhi:immlo, 64);
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <label> Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as "immhi:immlo" times 4096.

## Operation

```
bits(64) base = PC[];

base<11:0> = if page then
    base<11:0> = Zeros(12);

X[d] = base + imm;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = 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, -) = 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;

result = operand1 AND imm;
if d == 31 then 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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	0	shift	0	Rm						imm6						Rn						Rd			
opc								N																							

### 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;
if sf == '0' && imm6<5> == '1' then UNDEFINED; 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;

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

result = operand1 AND operand2; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 \(immediate\)](#).

31	30	29	28	27	26	25	24	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	0	N	immr						imms						Rn				Rd					
opc																															

### 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;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = 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, -) = 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;

result = operand1 AND imm;
PSTATE.<N,Z,C,V> = result<datasize-1>; 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):'00';[] = 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 \(shifted register\)](#).

31	30	29	28	27	26	25	24	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	0	shift	0	Rm						imm6						Rn						Rd							
opc								N																											

### 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;

if sf == '0' && imm6<5> == '1' then UNDEFINED; 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;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);(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
TST (shifted register)	Rd == '11111'

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>;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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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

(FEAT\_PAuth)

31	30	29	28	27	26	25	24	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;
```

### 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

```
auth_then_branch = FALSE;
if HavePACExt() then
    if source_is_sp then
        X[d] = AuthDA(X[d], SP[], FALSE);
    [], auth_then_branch);
else
    X[d] = AuthDA(X[d], X[n], FALSE); [n], auth_then_branch);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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

(FEAT\_PAuth)

31	30	29	28	27	26	25	24	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	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;
```

### 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

```
auth_then_branch = FALSE;
if HavePACExt() then
    if source_is_sp then
        X[d] = AuthDB(X[d], SP[], FALSE);
    [], auth_then_branch);
else
    X[d] = AuthDB(X[d], X[n], FALSE); [n], auth_then_branch);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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

(FEAT\_PAuth)

31	30	29	28	27	26	25	24	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	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;
```

### System

(FEAT\_PAuth)

31	30	29	28	27	26	25	24	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

```
auth_then_branch = FALSE;  
if HavePACExt() then  
  if source_is_sp then  
    X[d] = AuthIA(X[d], SP[], FALSE);  
[], auth_then_branch);  
  else  
    X[d] = AuthIA(X[d], X[n], FALSE);[n], auth_then_branch);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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

(FEAT\_PAuth)

31	30	29	28	27	26	25	24	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	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;
```

### System

(FEAT\_PAuth)

31	30	29	28	27	26	25	24	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";  
  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

```
auth_then_branch = FALSE;  
if HavePACExt() then  
  if source_is_sp then  
    X[d] = AuthIB(X[d], SP[], FALSE);  
[], auth_then_branch);  
  else  
    X[d] = AuthIB(X[d], X[n], FALSE);[n], auth_then_branch);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

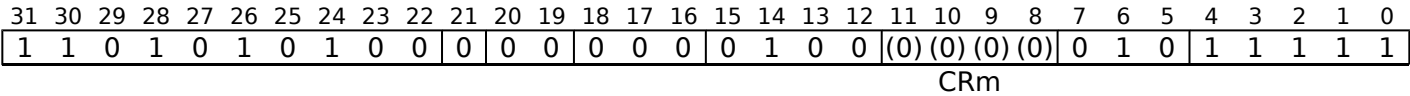
(new)



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  
(FEAT\_FlagM2)



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 = '0';
PSTATE.N = N;
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = '0'; PSTATE.V = V;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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						

B.<cond> <label>

```
bits(64) offset = SignExtend(imm19:'00', 64);(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(cond) then(condition) then
  BranchTo(PC[] + offset, BranchType_DIR);
```

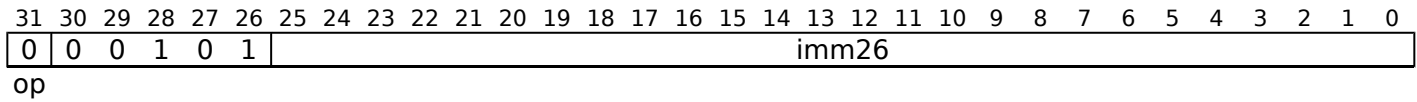
(old)

htmldiff from-

(new)

## 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.



B <label>

```
BranchType branch_type = if op == '1' then BranchType_DIRCALL else BranchType_DIR;
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_DIRCALL then
    X[30] = PC[] + 4;
BranchTo(PC[] + offset, BranchType_DIR);[] + offset, branch_type);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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 = UInt(immr);
(wmask, tmask) = S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

## 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;immr&gt;</code>	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.
<code>&lt;imms&gt;</code>	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
BFC	Rn == '11111' && UInt(imms) < UInt(immr)
BFI	Rn != '11111' && UInt(imms) < UInt(immr)
BFXIL	UInt(imms) >= UInt(immr)

Operation

```
bits(datasize) dst = 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, R) AND wmask;
(src<S>) else dst;

// combine extension bits and result bits
X[d] = (dst AND NOT(tmask)) OR (bot AND tmask); [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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	0	shift	1	Rm						imm6						Rn						Rd					
opc								N																									

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);
if invert then operand2 = NOT(operand2);

result = operand1 AND 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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	0	shift	1	Rm						imm6						Rn						Rd			
opc								N																							

### 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;

if sf == '0' && imm6<5> == '1' then UNDEFINED; 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;

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

operand2 = NOT(operand2);
if invert then operand2 = NOT(operand2);

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

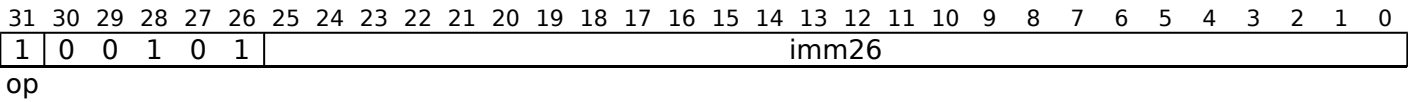
(old)

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.



BL <label>

```
BranchType branch_type = if op == '1' then BranchType_DIRCALL else BranchType_DIR;
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_DIRCALL then
    X[30] = PC[] + 4;
BranchTo(PC[] + offset, BranchType_DIRCALL);[] + offset, branch_type);
```

## 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										
Z							op				A				M		Rn							Rm							

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 && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == BranchType_RET 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];[n];
boolean auth_then_branch = TRUE;

if pac then
    bits(64) modifier = if source_is_sp then
SP[] else X[30] =[m];

    if use_key_a then
        target = AuthIA(target, modifier, auth_then_branch);
    else
        target = AuthIB(target, modifier, auth_then_branch);

if branch_type == BranchType_INDCALL then
    X[30] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10'; case branch_type of
    when
BranchToBranchType_INDIR(target, // BR, BRAA, BRAB, BRAAZ, BRABZ
    if InGuardedPage then
        if n == 16 || n == 17 then
            BTypeNext = '01';
        else
            BTypeNext = '11';
    else
        BTypeNext = '01';
    when BranchType_INDCALL // BLR, BLRAA, BLRAB, BLRAAZ, BLRABZ
        BTypeNext = '10';
    when BranchType_RET // RET, RETAA, RETAB
        BTypeNext = '00';

BranchTo);(target, branch_type);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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

(FEAT\_PAuth)

31	30	29	28	27	26	25	24	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	Z	0	0	1	1	1	1	1	1	0	0	0	0	1	M	Rn						Rm									
op																A																					

### 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);
integer m = (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 !if !pac && m != 0 then
    UNDEFINED;
elseif pac && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == BranchType_RET() then
        UNDEFINED;

if Z == '0' && m != 31 then
    UNDEFINED; 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];
boolean auth_then_branch = TRUE;

bits(64) modifier = if source_is_sp then if pac then
    bits(64) modifier = if source_is_sp then SP[] else X[m];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = (target, modifier, auth_then_branch);
    else
        target = AuthIB(target, modifier, TRUE); (target, modifier, auth_then_branch);

if branch_type ==
BranchType_INDICAL then
    X[30] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10'; case branch_type of
    when
BranchToBranchType_INDIRE(target, // BR, BRAA, BRAB, BRAAZ, BRABZ
    if InGuardedPage then
        if n == 16 || n == 17 then
            BTypeNext = '01';
        else
            BTypeNext = '11';
        else
            BTypeNext = '01';
    when BranchType_INDICAL // BLR, BLRAA, BLRAB, BLRAAZ, BLRABZ
        BTypeNext = '10';
    when BranchType_RET // RET, RETAA, RETAB
        BTypeNext = '00';

BranchTo); (target, branch_type);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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		Rn							Rm									

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 && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == BranchType_RET 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];
boolean auth_then_branch = TRUE;

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01'; if pac then
        bits(64) modifier = if source_is_sp then
BranchToSP(target,[]) else X[m];

    if use_key_a then
        target = AuthIA(target, modifier, auth_then_branch);
    else
        target = AuthIB(target, modifier, auth_then_branch);

if branch_type == BranchType_INDCALL then
    X[30] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
case branch_type of
    when BranchType_INDIR // BR, BRAA, BRAB, BRAAZ, BRABZ
        if InGuardedPage then
            if n == 16 || n == 17 then
                BTypeNext = '01';
            else
                BTypeNext = '11';
        else
            BTypeNext = '01';
    when BranchType_INDCALL // BLR, BLRAA, BLRAB, BLRAAZ, BLRABZ
        BTypeNext = '10';
    when BranchType_RET // RET, RETAA, RETAB
        BTypeNext = '00';

BranchTo);(target, branch_type);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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

(FEAT\_PAuth)

31	30	29	28	27	26	25	24	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	Z	0	0	0	1	1	1	1	1	0	0	0	0	1	M	Rn						Rm									
op																A																					

### 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);
integer m = (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 !if !pac && m != 0 then
    UNDEFINED;
elseif pac && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == BranchType_RET() then
        UNDEFINED;

if Z == '0' && m != 31 then
    UNDEFINED; 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];
boolean auth_then_branch = TRUE;

bits(64) modifier = if source_is_sp then if pac then
    bits(64) modifier = if source_is_sp then SP[] else X[m];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = (target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, TRUE);
(target, modifier, auth_then_branch);

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01'; if branch_type ==
BranchToBranchType_INDCALL (target, then X[30] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
case branch_type of
    when BranchType_INDIR // BR, BRAA, BRAB, BRAAZ, BRABZ
        if InGuardedPage then
            if n == 16 || n == 17 then
                BTypeNext = '01';
            else
                BTypeNext = '11';
        else
            BTypeNext = '01';
    when BranchType_INDCALL // BLR, BLRAA, BLRAB, BLRAAZ, BLRABZ
        BTypeNext = '10';
    when BranchType_RET // RET, RETAA, RETAB
        BTypeNext = '00';

BranchTo); (target, branch_type);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

BRK

Breakpoint instruction. A 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.

31	30	29	28	27	26	25	24	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	1	imm16																0	0	0	0	0

```
BRK #<imm>

bits(16) comment = imm16;
if HaveBTIExt() then
    SetBTypeCompatible(TRUE);
```

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.SoftwareBreakpoint(imm16);(comment);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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*.BTTYPE != 0b00, a BTI instruction compatible with the current value of *PSTATE*.BTTYPE 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*.BTTYPE which the BTI instruction is compatible with.

Within a guarded memory region, while *PSTATE*.BTTYPE != 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  
(FEAT\_BTI)

31	30	29	28	27	26	25	24	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	x	x	0	1	1	1	1	1				
																					CRm				op2											

BTI {<targets>}

SystemHintOp op;

```
if CRm.op2 == '0100 xx0' then
  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 110'
    if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_DGH;
  when '0000 111' SEE "XPACLRI";
  when '0001 xxx'
    case op2 of
    when '000' SEE "PACIA1716";
    when '010' SEE "PACIB1716";
    when '100' SEE "AUTIA1716";
    when '110' SEE "AUTIB1716";
    otherwise EndOfInstruction(); // Instruction executes as NOP
  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'
    case op2 of
    when '000' SEE "PACIAZ";
    when '001' SEE "PACIASP";
    when '010' SEE "PACIBZ";
    when '011' SEE "PACIBSP";
    when '100' SEE "AUTIAZ";
    when '101' SEE "AUTHASP";
    when '110' SEE "AUTIBZ";
    when '111' SEE "AUTIBSP";
  when '0100 xx0'
    op = SystemHintOp_BTI;
  // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
  SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
else 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_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(-1, (localtimeout, WFXType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(-1, (localtimeout, WFXType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext('00');

  otherwise // do nothing
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = 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;
boolean tag_checked = n != 31;
```

## 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(8) comparevalue;
bits(8) newvalue;
bits(8) data;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

X[s] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341  
Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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	Rn				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 = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);{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;
boolean tag_checked = n != 31;
```

## 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(16) comparevalue;
bits(16) newvalue;
bits(16) data;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);
X[s] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341  
Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	sz	0	0	1	0	0	0	0	L	1	Rs			o0	1	1	1	1	1	Rn			Rt								
Rt2																															

Rt2

### 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 = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 32 << UInt(sz);(sz);
integer regsize = datasize;
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

```

## 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(ldacctype) then s1:s2 else s2:s1;
newvalue = if BigEndian(stacctype) then t1:t2 else t2:t1;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

if BigEndian(ldacctype) then
    X[s] = data<2*datasize-1:datasize>;
    X[s+1] = data<datasize-1:0>;
else
    X[s] = data<datasize-1:0>;
    X[s+1] = data<2*datasize-1:datasize>;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

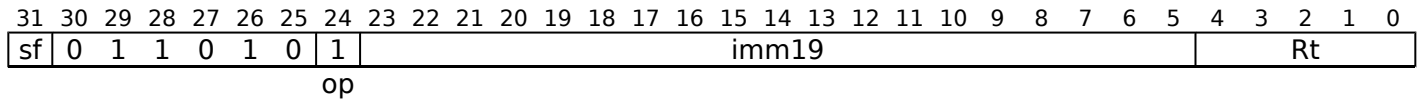
(old)

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) == FALSE then (operand1) == iszero then
  BranchTo(PC[] + offset, BranchType_DIR);
```

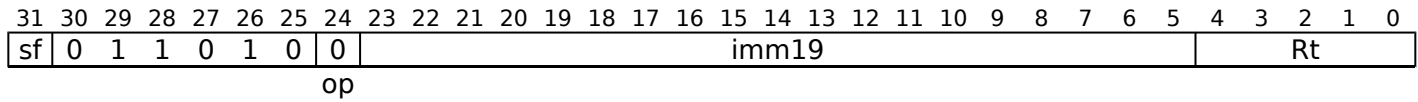
Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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) == TRUE then(operand1) == iszero then
  BranchTo(PC[] + offset, BranchType_DIR);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CCMN (immediate)

Conditional Compare Negative (immediate) sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	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	1	0	1	0	0	1	0	imm5					cond					1	0	Rn					0	nzcw			
op																																

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

CCMN <Wn>, #<imm>, #<nzcw>, <cond>

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

CCMN <Xn>, #<imm>, #<nzcw>, <cond>

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcw;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

## Assembler Symbols

- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <imm> Is a five bit unsigned (positive) immediate encoded in the "imm5" field.
- <nzcw> 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 "nzcw" field.
- <cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
if ConditionHolds(cond) then
  (condition) then
    bits(datasize) operand1 = X[n];
    bits(datasize) operand2 = imm;
    bit carry_in = '0';
    if sub_op then
      operand2 = NOT(operand2);
      carry_in = '1';
    (-, flags) = AddWithCarry(operand1, imm, '0');
    (operand1, operand2, carry_in);
    PSTATE.<N,Z,C,V> = flags;
```

## 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)

## CCMN (register)

Conditional Compare Negative (register) sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	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	1	0	1	0	0	1	0	Rm				cond				0	0	Rn				0	nzcv					
op																															

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

CCMN <Wn>, <Wm>, #<nzcv>, <cond>

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

CCMN <Xn>, <Xm>, #<nzcv>, <cond>

```
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

## Assembler Symbols

- <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.
- <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.
- <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.

## Operation

```
if ConditionHolds(cond) then
(condition) then
    bits(datasize) operand1 = X[n];
    bits(datasize) operand2 = X[m];
    bit carry_in = '0';
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, '0');
(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

## 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)

## CCMP (immediate)

Conditional Compare (immediate) sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	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	1	0	1	0	0	1	0	imm5					cond					1	0	Rn					0	nzcw		
op																															

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

CCMP <Wn>, #<imm>, #<nzcw>, <cond>

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

CCMP <Xn>, #<imm>, #<nzcw>, <cond>

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcw;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

## Assembler Symbols

- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <imm> Is a five bit unsigned (positive) immediate encoded in the "imm5" field.
- <nzcw> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCW condition flags, encoded in the "nzcw" field.
- <cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
if ConditionHolds(cond) then
  (condition) then
    bits(datasize) operand1 = X[n];
    bits(datasize) operand2;
    operand2 = NOT(imm);
    bits(datasize) operand2 = imm;
    bit carry_in = '0';
    if sub_op then
      operand2 = NOT(operand2);
      carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, '1');
    (operand1, operand2, carry_in);
    PSTATE.<N,Z,C,V> = flags;
```

## 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 NZCW 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 NZCW flags.

(old)

htmldiff from-

(new)

## CCMP (register)

Conditional Compare (register) sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.

31	30	29	28	27	26	25	24	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	1	0	1	0	0	1	0	Rm				cond				0	0	Rn				0	nzcv					
op																															

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

CCMP <Wn>, <Wm>, #<nzcv>, <cond>

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

CCMP <Xn>, <Xm>, #<nzcv>, <cond>

```
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzcv;
```

## Assembler Symbols

- <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.
- <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.
- <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.

## Operation

```
if ConditionHolds(cond) then
(condition) then
    bits(datasize) operand1 = X[n];
    bits(datasize) operand2 = X[m];
    operand2 = NOT(operand2);
    bit carry_in = '0';
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

## 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## CLS

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];

result = 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CLZ

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.

31	30	29	28	27	26	25	24	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	1	0	0	Rn					Rd				
op																															

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

CLZ &lt;Wd&gt;, &lt;Wn&gt;

### 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_ROR
```

## 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];

result = 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14Z

Copyright © 2010-2020 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 = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);(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 = 0x04C11DB7<31:0>;
[m]; // input value
bits(32) poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;

bits(32+size) tempacc = BitReverse(acc):(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);(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 = 0x1EDC6F41<31:0>;
[m]; // input value
bits(32) poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;

bits(32+size) tempacc = BitReverse(acc):(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 predication 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											



```

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 110'
    if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_DGH;
  when '0000 111' SEE "XPACLRI";
  when '0001 xxx'
    case op2 of
      when '000' SEE "PACIA1716";
      when '010' SEE "PACIB1716";
      when '100' SEE "AUTIA1716";
      when '110' SEE "AUTIB1716";
      otherwise EndOfInstruction(); // Instruction executes as NOP
    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'
      case op2 of
        when '000' SEE "PACIAZ";
        when '001' SEE "PACIASP";
        when '010' SEE "PACIBZ";
        when '011' SEE "PACIBSP";
        when '100' SEE "AUTIAZ";
        when '101' SEE "AUTHASP";
        when '110' SEE "AUTIBZ";
        when '111' SEE "AUTIBSP";
      when '0100 xx0'
        op = SystemHintOp_BTI;
        // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
        SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
      otherwise EndOfInstruction// Empty.(); // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp_YIELDHint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(localtimeout, WFXType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(localtimeout, WFXType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext();('00');

  otherwise // do nothing
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## CSEL

If the condition is true, Conditional Select writes the value of the first source register to the destination register. If the condition is false, it writes the value of the second source register 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	1	0	1	0	1	0	0	Rm				cond				0	0	Rn				Rd						
op											o2																				

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

CSEL <Wd>, <Wn>, <Wm>, <cond>

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

CSEL <Xd>, <Xn>, <Xm>, <cond>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '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.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
(condition) then
    result = X[n];
else
    result = X[m];[m];
if else_inv then result = NOT(result);
if else_inc then result = result + 1;
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.

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

## CSINC

Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases [CINC](#), and [CSET](#).

31	30	29	28	27	26	25	24	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	0	0	Rm					cond					0	1	Rn					Rd				
op											o2																					

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

CSINC <Wd>, <Wn>, <Wm>, <cond>

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

CSINC <Xd>, <Xn>, <Xm>, <cond>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '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.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Alias Conditions

Alias	Is preferred when
<a href="#">CINC</a>	Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm
<a href="#">CSET</a>	Rm == '11111' && cond != '111x' && Rn == '11111'

## Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
(condition) then
    result = X[n];
else
    result = X[m];
    result = result + 1; if else_inv then result = NOT(result);
    if else_inc then result = result + 1;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## CSINV

Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases [CINV](#), and [CSETM](#).

31	30	29	28	27	26	25	24	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	0	0	Rm					cond					0	0	Rn					Rd				
op											o2																					

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

CSINV <Wd>, <Wn>, <Wm>, <cond>

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

CSINV <Xd>, <Xn>, <Xm>, <cond>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '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.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Alias Conditions

Alias	Is preferred when
<a href="#">CINV</a>	Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm
<a href="#">CSETM</a>	Rm == '11111' && cond != '111x' && Rn == '11111'

## Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
(condition) then
    result = X[n];
else
    result = X[m];
    result = NOT(result); if else_inv then result = NOT(result);
    if else_inc then result = result + 1;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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



## CSNEG

Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias [CNEG](#).

31	30	29	28	27	26	25	24	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	0	0	Rm					cond					0	1	Rn					Rd				
op											o2																					

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

CSNEG <Wd>, <Wn>, <Wm>, <cond>

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

CSNEG <Xd>, <Xn>, <Xm>, <cond>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
bits(4) condition = cond;
boolean else_inv = (op == '1');
boolean else_inc = (o2 == '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.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Alias Conditions

Alias	Is preferred when
<a href="#">CNEG</a>	cond != '111x' && Rn == Rm

## Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
  (condition) then
    result = X[n];
  else
    result = X[m];
  result = NOT(result);
  result = result + 1; if else_inv then result = NOT(result);
  if else_inc then result = result + 1;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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*.

31	30	29	28	27	26	25	24	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	1	0	1	imm16																0	0	0	0	1
																												LL			

DCPS1 {#<imm>}

```
bits(2) target_level = LL;
if LL == '00' then UNDEFINED;
if !Halted() then UNDEFINED;
```

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(LL);(target_level);
```

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 if EL2 is disabled in the current Security state.

This instruction is always UNDEFINED in Non-debug state.  
For more information on the operation of the DCPSn instructions, see [DCPS](#).

31	30	29	28	27	26	25	24	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	1	0	1	imm16															0	0	0	1	0	
																												LL			

DCPS2 {#<imm>}

```
bits(2) target_level = LL;
if LL == '00' then UNDEFINED;
if !Halted() then UNDEFINED;
```

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(LL);(target_level);
```

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*.

31	30	29	28	27	26	25	24	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	1	0	1	imm16																0	0	0	1	1
LL																															

DCPS3 {#<imm>}

```
bits(2) target_level = LL;
if LL == '00' then UNDEFINED;
if !Halted() then UNDEFINED;
```

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(LL);(target_level);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## DGH

DGH is a hint instruction. A DGH instruction is not expected to be performance optimal to merge memory accesses with Normal Non-cacheable or Device-GRE attributes appearing in program order before the hint instruction with any memory accesses appearing after the hint instruction into a single memory transaction on an interconnect.

### System (FEAT\_DGH)

31	30	29	28	27	26	25	24	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	1	0	1	1	1	1	1
																CRm				op2											

DGH

```

SystemHintOp if !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 110'
        if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_DGH;
    when '0000 111' SEE "XPACLR1";
    when '0001 xxx'
        case op2 of
            when '000' SEE "PACIA1716";
            when '010' SEE "PACIB1716";
            when '100' SEE "AUTIA1716";
            when '110' SEE "AUTIB1716";
            otherwise EndOfInstruction(); // Instruction executes as NOP
        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'
            case op2 of
                when '000' SEE "PACIAZ";
                when '001' SEE "PACIASP";
                when '010' SEE "PACIBZ";
                when '011' SEE "PACIBSP";
                when '100' SEE "AUTIAZ";
                when '101' SEE "AUTHASP";
                when '110' SEE "AUTIBZ";
                when '111' SEE "AUTIBSP";
            when '0100 xx0'
                op = SystemHintOp_BTI;
                // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
                SetBTypeCompatible(BTypeCompatible_BTI() then (op2 < 2:1 >));
            otherwise EndOfInstruction(); // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp_YIELDHint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(localtimeout, WFEType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(localtimeout, WFIType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext();('00');

  otherwise // do nothing
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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.

31	30	29	28	27	26	25	24	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		0		shift		1		Rm						imm6						Rn						Rd			
opc																N																									

### 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;
if sf == '0' && imm6<5> == '1' then UNDEFINED; 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;

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

operand2 = NOT(operand2);
if invert then operand2 = NOT(operand2);

result = operand1 EOR 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

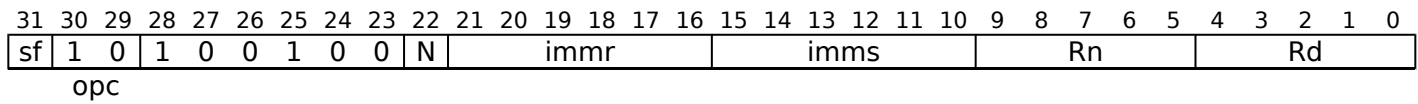
(old)

htmldiff from-

(new)

## 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.



### 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;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) =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, -) = 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;

result = operand1 EOR imm;

if d == 31 then 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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 0		shift		0		Rm				imm6				Rn				Rd							
opc								N																							

### 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;
if sf == '0' && imm6<5> == '1' then UNDEFINED; 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;

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

result = operand1 EOR operand2; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

(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			

ERET

```

if PSTATE.EL == EL0 then UNDEFINED;then UNDEFINED;
boolean pac = (A == '1');
boolean use_key_a = (M == '0');

if !pac && op4 != '00000' then
    UNDEFINED;
elseif pac && (!HavePACExt() || op4 != '11111') then
    UNDEFINED;

if Rn != '11111' then
    UNDEFINED;

```

## Operation

```

AArch64.CheckForERetTrap(FALSE, TRUE);
(pac, use_key_a);
bits(64) target = ELR[];{};
boolean auth_then_branch = TRUE;

if pac then
    if use_key_a then
        target =

AuthIA(ELR[], SP[], auth_then_branch);
    else
        target = AuthIB(ELR[], SP[], auth_then_branch);

AArch64.ExceptionReturn(target, SPSR[]);

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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

(FEAT\_PAuth)

31	30	29	28	27	26	25	24	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 !if !pac && op4 != '00000' then
    UNDEFINED;
elseif pac && (!HavePACExt() then
    () || op4 != '11111') then
    UNDEFINED;

if Rn != '11111' then
    UNDEFINED;

```

### Operation

```

AArch64.CheckForERetTrap(TRUE, use_key_a);
bits(64) target;

if use_key_a then
    target = {pac, use_key_a};
bits(64) target = ELR[];
boolean auth_then_branch = TRUE;

if pac then
    if use_key_a then
        target = AuthIA(ELR[], SP[], TRUE);
    else
        target = [], auth_then_branch;
    else
        target = AuthIB(ELR[], SP[], TRUE); [], auth_then_branch;

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  
(FEAT\_RAS)

31	30	29	28	27	26	25	24	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											

```

SystemHintOp if !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 110'
    if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_DGH;
  when '0000 111' SEE "XPACLR1";
  when '0001 xxx'
    case op2 of
      when '000' SEE "PACIA1716";
      when '010' SEE "PACIB1716";
      when '100' SEE "AUTIA1716";
      when '110' SEE "AUTIB1716";
      otherwise EndOfInstruction(); // Instruction executes as NOP
    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'
      case op2 of
        when '000' SEE "PACIAZ";
        when '001' SEE "PACIASP";
        when '010' SEE "PACIBZ";
        when '011' SEE "PACIBSP";
        when '100' SEE "AUTIAZ";
        when '101' SEE "AUTHASP";
        when '110' SEE "AUTIBZ";
        when '111' SEE "AUTIBSP";
      when '0100 xx0'
        op = SystemHintOp_BTI;
        // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
        SetBTypeCompatible(BTypeCompatible_BTI() then (op2<2:1>));
      otherwise EndOfInstruction(); // Instruction executes as NOP
    end
  end
end

```

## Operation

```
if case op of
  when SystemHintOp_YIELDHint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(localtimeout, WFEType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(localtimeout, WFIType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
    SynchronizeErrors();
    AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext(); ('00');

  otherwise // do nothing
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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

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 110'
```

```
    if !HaveDGHExt() then EndOfInstruction();    // Instruction executes as NOP
    op = SystemHintOp\_DGH;
```

```
when '0000 111' SEE "XPACLR1";
```

```
when '0001 xxx'
```

```
    case op2 of
```

```
        when '000' SEE "PACIA1716";
        when '010' SEE "PACIB1716";
        when '100' SEE "AUTIA1716";
        when '110' SEE "AUTIB1716";
        otherwise EndOfInstruction();
```

```
    ();    // Instruction executes as NOP
```

```
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'
```

```
    case op2 of
```

```
        when '000' SEE "PACIAZ";
        when '001' SEE "PACIASP";
        when '010' SEE "PACIBZ";
        when '011' SEE "PACIBSP";
        when '100' SEE "AUTIAZ";
        when '101' SEE "AUTHASP";
        when '110' SEE "AUTIBZ";
        when '111' SEE "AUTIBSP";
```

```
when '0100 xx0'
```

```
    op = SystemHintOp\_BTI;
```

```
    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
```

```
    SetBTypeCompatible(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_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(-1, (localtimeout, WFXType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(-1, (localtimeout, WFXType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext('00');

  otherwise // do nothing
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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.
- At EL1 if EL2 is not enabled in the current Security state.
- 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

HVC #<imm>

```
// Empty.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 HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);

if hvc_enable == '0' then
    UNDEFINED;
else
    AArch64.CallHypervisor(imm16);(imm);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD64B

Single-copy Atomic 64-byte Load derives an address from a base register value, loads eight 64-bit doublewords from a memory location, and writes them to consecutive registers, Xt to X(t+7). The data that is loaded is atomic and is required to be 64-byte aligned.

### Integer (FEAT\_LS64)

31	30	29	28	27	26	25	24	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	0	0	1	1	1	1	1	1	1	1	0	1	0	0	Rn				Rt					

LD64B <Xt>, [<Xn|SP> {, #0}]

```

if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

```

### Assembler Symbols

- <Xt> Is the 64-bit name of the first 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

```

CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
acctype = AccType_ATOMICS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemLoad64B(address, acctype);

for i = 0 to 7
    value = data<63+64*i:64*i>;
    value = data<63+64*i : 64*i>;
    if BigEndian(acctype) then value = BigEndianReverse(value);
    X[t+i] = value;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	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	0	0	0	0	0	Rn				Rt						
size											opc																				



### 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 = 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_ATOMICRW;  
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;  
boolean tag_checked = n != 31; MemAtomicOp op;  
case op 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;  
boolean tag_checked = n != 31;
```

## 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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});
if t != 31 then MemAtomicOp_ADD, value, ldacctype, stacctype);
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = 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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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(16) value;
bits(16) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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

(FEAT\_LRCPC)

31	30	29	28	27	26	25	24	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	1	0	1	(1)	(1)	(1)	(1)	(1)	1	1	0	0	0	0	Rn				Rt					
size											Rs																				

#### 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 elsize = 8 << integer s = UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31; (Rs); // ignored by all loads and store-release
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
boolean tag_checked = n != 31;
```

### 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(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, dbytes, address, dbytes, acctype]; AccType_ORDERED];
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 (FEAT\_LRCPC)

31	30	29	28	27	26	25	24	23	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	0	1	(1)	(1)	(1)	(1)	(1)	1	1	0	0	0	0	Rn					Rt					
size											Rs																					

LDAPRB <Wt>, [<Xn|SP> {, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;
```

### 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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1, [address, dbytes, acctype], AccType_ORDERED];
X[t] = ZeroExtend(data, 32); (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 (FEAT\_LRCPC)

31	30	29	28	27	26	25	24	23	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	0	1	(1)	(1)	(1)	(1)	(1)	1	1	0	0	0	0	Rn					Rt					
size											Rs																					

LDAPRH <Wt>, [<Xn|SP> {, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;
```

### 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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 2, [address, dbytes, acctype], AccType_ORDERED];
X[t] = ZeroExtend(data, 32); (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](#).

### Unscaled offset (FEAT\_LRCPC2)

31	30	29	28	27	26	25	24	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);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype}] = data;

    when AccType_ORDEREDMemOp_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(data, regsize)[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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](#).

### Unscaled offset (FEAT\_LRCPC2)

31	30	29	28	27	26	25	24	23	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																									

```
LDAPURB <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset =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);

boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, {address, datasize DIV 8, acctype}] = data;

    when AccType_ORDEREDMemOp_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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32)[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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](#).

### Unscaled offset (FEAT\_LRCPC2)

31	30	29	28	27	26	25	24	23	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																								

LDAPURH <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset = 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);

boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, {address, datasize DIV 8, acctype}] = data;

    when AccType_ORDEREDMemOp_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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32)[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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](#).

### Unscaled offset (FEAT\_LRCPC2)

31	30	29	28	27	26	25	24	23	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>{, #<sim>}]
```

### 64-bit (opc == 10)

```
LDAPURSB <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = 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_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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction && (n != 31);();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, [address, datasize DIV 8, acctype]] = data;

    when AccType_ORDERED] = data;

    when MemOp_LOAD
        data = Mem[address, 1, [address, datasize DIV 8, acctype]];
        if signed then AccType_ORDERED];
        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(address, t<4:0>); [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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](#).

### Unscaled offset (FEAT\_LRCPC2)

31	30	29	28	27	26	25	24	23	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	1	x	0	imm9									0	0	Rn				Rt									
size										opc																									

### 32-bit (opc == 11)

```
LDAPURSH <Wt>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (opc == 10)

```
LDAPURSH <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = 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_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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction && (n != 31);();
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, [address, datasize DIV 8, acctype] = data;

    when AccType_ORDERED] = data;

    when MemOp_LOAD
        data = Mem[address, 2, [address, datasize DIV 8, acctype];
        if signed then AccType_ORDERED];
        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(address, t<4:0>);[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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](#).

### Unscaled offset (FEAT\_LRCPC2)

31	30	29	28	27	26	25	24	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																									

LDAPURSW <Xt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = 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.
- <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);

boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 4, {address, datasize DIV 8, acctype}] = data;

    when AccType_ORDEREDMemOp_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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 64)[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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/XZR, 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 elsize = 8 << integer t2 = UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31; (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;
boolean tag_checked = n != 31;
```

## 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(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, [address, dbytes, acctype] = data;

    when
        data = MemAccType_ORDEREDMemOp_LOAD]; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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/XZR, 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														

LDARB <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;
```

## 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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, [address, dbytes, acctype] = data;

    when
        data = MemAccType_ORDEREDMemOp_LOAD; [address, dbytes, acctype];
X[t] = ZeroExtend(data, 32); (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/XZR, 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	1	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														

LDARH <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;
```

## 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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, [address, dbytes, acctype] = data;

    when
        data = MemAccType_ORDEREDMemOp_LOAD; [address, dbytes, acctype];
X[t] = ZeroExtend(data, 32); (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);

integer elsize = 32 << (Rt2); // ignored by load/store single register
integer s = UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then (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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 rt_unknown = TRUE; // result is UNKNOWN
      when rn_unknown = FALSE; // address is original base
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
```

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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elseif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// 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.
when
    AArch64.SetExclusiveMonitorsMemOp_STORE(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    data = bits(datasize) UNKNOWN;
    elseif pair then
        bits(datasize DIV 2) el1 =
            X[t] = bits(datasize) UNKNOWN; // In this case t = t2
    elseif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = {t};
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
    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, [address, dbytes, acctype]] = data;
        status = AccType_ORDEREDATOMICExclusiveMonitorsStatus;
    if(); BigEndianX([s] = AccType_ORDEREDATOMICZeroExtend) then(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] = data<datasize-1:elsize>; [t] = bits(datasize) UNKNOWN; // In this case t =
        elseif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data =
                Mem[address, dbytes, acctype];
            if BigEndian(acctype) then
                X[t2] = data<elsize-1:0>;
            else [t] = data<datasize-1:elsize>;
            X[t] = data<elsize-1:0>; [t2] = data<elsize-1:0>;
        else
            X[t2] = data<datasize-1:elsize>;

```

```

else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != [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 (address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault((acctype, iswrite, secondstage)); AccType_ORDEREDATOMIC);
            Mem[address + 0, 8, acctype];
            X[t] = [t2] = Mem[address, 8, [address + 8, 8, acctype]];
        else
            data = AccType_ORDEREDATOMICMem; [address, dbytes, acctype];
            X[t2] = [t] = MemZeroExtend[address+8, 8, AccType_ORDEREDATOMIC]; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14.5341

Copyright © 2010-2020 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 elsize = 8 << integer t2 = UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31; (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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 UNDEFINED;
            when Constraint_NOP EndOfInstruction();
```

## 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(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// 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.
when
    MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
        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);

data =
    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN; // In this case t
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, {address, dbytes, acctype}];
            if (acctype) 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 = MemAccType_ORDEREDATOMICBigEndian[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53.41

Copyright © 2010-2020 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													

LDAXRB <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 UNDEFINED;
            when Constraint_NOP EndOfInstruction();
```

## 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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// 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.
when
    MemOp_STORE
    if rt_unknown then
        data = bits(datasize) UNKNOWN;
    elsif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
    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, 1);
(address, dbytes);

data = if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN; // In this case t
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, 1, [address, dbytes, acctype]];
        if (acctype) 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 = MemAccType_ORDEREDATOMICBigEndian[address, dbytes, acctype];
    X[t] = ZeroExtend(data, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14Z5341

Copyright © 2010-2020 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														

LDAXRH <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 UNDEFINED;
            when Constraint_NOP EndOfInstruction();
```

## 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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// 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.
when
    MemOp_STORE
    if rt_unknown then
        data = bits(datasize) UNKNOWN;
    elsif pair then
        bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
    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, 2);
(address, dbytes);

data = if pair then
    if rt_unknown then
        // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN; // In this case t
    elsif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = Mem[address, 2, [address, dbytes, acctype]];
        if (acctype) 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 = MemAccType_ORDEREDATOMICBigEndian[address, dbytes, acctype];
    X[t] = ZeroExtend(data, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14.5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	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	0	0	1	0	0	Rn				Rt						
size											opc																				



### 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 = 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_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31; ;MemAtomicOp_op;
case op 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;
boolean tag_checked = n != 31;
```

## 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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});
if t != 31 then MemAtomicOp_BIC, value, ldacctype, stacctype);
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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(8) value;
bits(8) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});
if t != 31 then MemAtomicOp_BIC, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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="#">STCLRHL, STCLRLH</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	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	0	1	0	0	0	Rn				Rt						
size											opc																				

### 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 = 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_ATOMICRW;
```

```
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

```
boolean tag_checked = n != 31; ;MemAtomicOp_op;
```

```
case op 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;
```

```
boolean tag_checked = n != 31;
```

## 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, STEORL</a>	A == '0' && Rt == '11111'

Operation

```

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});
if t != 31 then MemAtomicOp_EOR, value, ldacctype, stacctype);
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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(8) value;
bits(8) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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(16) value;
bits(16) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+00:00

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## LDGM

Load Tag Multiple reads a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID\_EL1.BS, and writes the Allocation Tag read from address A to the destination register at  $4*A < 7:4 > + 3:4*A < 7:4 >$ . Bits of the destination register not written with an Allocation Tag are set to 0.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

If `ID_AA64PFR1_EL1.MTE` != 0b0010, this instruction is UNDEFINED.

### Integer (FEAT\_MTE2)

31	30	29	28	27	26	25	24	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					

LDGM <Xt>, [<Xn|SP>]

```
if !HaveMTE2Ext() then UNDEFINED;
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

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = Zeros(64);
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

integer size = 4 * (2 ^ (integer size = 4*(2^(UInt(GMID_EL1.BS))));
address = Align(address, size);
(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);

for i = 0 to count-1
    bits(4) tag = AArch64.MemTag[address, AccType_NORMAL];
    data<(index*4)+3:index*4> = tag;
    address = address + TAG_GRANULE;
    index = index + 1;

X[t] = data;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

### No offset (FEAT\_LOR)

31	30	29	28	27	26	25	24	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)	0	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size								L			Rs					o0		Rt2													

### 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 elsize = 8 << integer t2 = UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31; (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;
boolean tag_checked = n != 31;
```

## 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(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, dbytes, [address, dbytes, acctype] = data;

    when
        data = MemAccType_LIMITEDORDEREDMemOp_LOAD];[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

### No offset (FEAT\_LOR)

31	30	29	28	27	26	25	24	23	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															

LDLARB <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;
```

### 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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, [address, dbytes, acctype]] = data;

    when
        data = MemAccType_LIMITEDORDEREDMemOp_LOAD; [address, dbytes, acctype];
X[t] = ZeroExtend(data, 32); (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

### No offset (FEAT\_LOR)

31	30	29	28	27	26	25	24	23	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														

```
LDLARH <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;
```

### 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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, [address, dbytes, acctype]] = data;

    when
        data = MemAccType_LIMITEDORDEREDMemOp_LOAD; [address, dbytes, acctype];
X[t] = ZeroExtend(data, 32); (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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>}]
```

```
// Empty.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);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + (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 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if memop == MemOp_LOAD(imm7, 64), scale);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then && 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 Constraint_NOP     EndOfInstruction();
```

## Operation

```

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data1 = 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, dbytes, [address + 0, dbytes, acctype] = data1; AccType_STREAMMem];
data2 = [address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype];
        data2 = AccType_STREAMMem];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN; [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[t] = data1; [] = address;
    else
        X[t2] = data2; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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> 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.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + (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 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then if memop ==
MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
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 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 Constraint_NOP EndOfInstruction();

if memop == MemOp_LOAD();

if t == t2 then && 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 Constraint_NOP EndOfInstruction();

```

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    if ! postindex then
        address = address + offset;

data1 = 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, dbytes, [address + 0, dbytes, acctype] = data1; AccType_NORMALMem];
data2 = [address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype];
        data2 = AccType_NORMALMem];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if signed then [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																					

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																					

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																					

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);
bits(64) offset = (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 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), 2);
(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then if memop ==
MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
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 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 Constraint_NOP EndOfInstruction();

if memop == MemOp_LOAD();

if t == t2 then && 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 Constraint_NOP EndOfInstruction();

```

## Operation

```
bits(64) address;
bits(32) data1;
bits(32) data2;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    if !postindex then
        address = address + offset;

data1 = 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, 4, [address + 0, dbytes, acctype] = data1; AccType_NORMALMem];
data2 = [address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address+4, 4, [address + 0, dbytes, acctype];
        data2 = AccType_NORMALMem];
if rt_unknown then
    data1 = bits(32) UNKNOWN;
    data2 = bits(32) UNKNOWN; [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;
[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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.
<sim>	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);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 wback = FALSE; // writeback is suppressed
rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN    wb_unknown = TRUE; // writeback is UNKNOWN
rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

if !postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMALMemOp_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(data, regsize);
(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

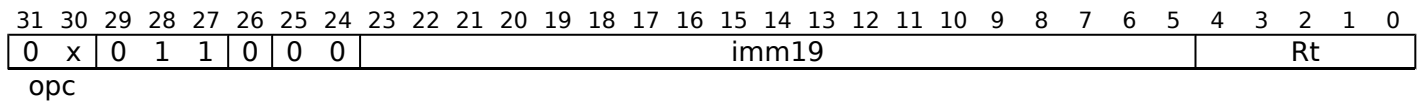
(old)

htmldiff from-

(new)

## LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



### 32-bit (opc == 00)

```
LDR <Wt>, <label>
```

### 64-bit (opc == 01)

```
LDR <Xt>, <label>
```

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 4;
    signed = TRUE;
  when '11'
    memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);(imm19:'00', 64);
boolean tag_checked = FALSE;
```

## 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.
- <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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
(tag_checked);

case memop of
    when MemOp_LOAD
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCHPrefetch(address, t<4:0>);
```

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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	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	1	Rm				option		S	1	0	Rn				Rt							
size											opc																				

### 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 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);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then if n == 31 then
    if memop !=
        MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype] = data;

    when AccType_NORMALMemOp_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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, regsize); [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 (FEAT\_PAuth)

31	30	29	28	27	26	25	24	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() then UNDEFINED;
() || size != '11' then UNDEFINED;
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), 3);
(S10, 64), scale);
boolean tag_checked = wback || n != 31;
```

### 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;
boolean auth_then_branch = TRUE;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

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 Constraint_NOP        EndOfInstruction();

if n == 31 then
    address = SP[];
else
    address = X[n];

if use_key_a then
    address = AuthDA(address, X[31], TRUE);
[31], auth_then_branch);
else
    address = AuthDB(address, X[31], TRUE);
[31], auth_then_branch);

if n == 31 then
    CheckSPAlignment();

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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																									

LDRB <Wt>, [<Xn|SP>], #<sim>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = 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																																

LDRB <Wt>, [<Xn|SP>, #<sim>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = 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																												

LDRB <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);(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);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPD);
    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 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 wback = FALSE; // writeback is suppressed
rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

if !postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
    Mem[address, 1, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMALMemOp_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(data, 32);
(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

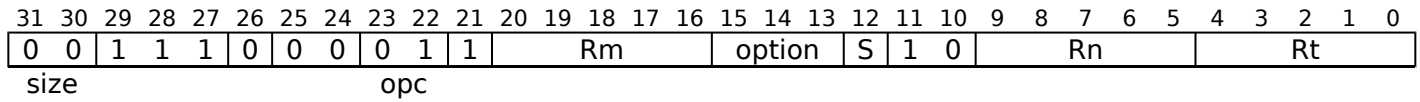
(old)

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>}]

```
if option<1> == '0' then UNDEFINED; // sub-word index
boolean postindex = FALSE;
integer scale =
  UInt(size);
if option<1> == '0' then UNDEFINED; // 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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, 0);
(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, [address, datasize DIV 8, acctype] = data;

    when AccType_NORMAL 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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32); [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																									

LDRH <Wt>, [<Xn|SP>], #<sim>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = 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																																

LDRH <Wt>, [<Xn|SP>, #<sim>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = 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	1	0	0		1	0		1	imm12								Rn				Rt									
size										opc																									

LDRH <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);(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);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPD);
    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 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 wback = FALSE; // writeback is suppressed
rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

if !postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMALMemOp_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(data, 32);
    (address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elseif 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	23	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	1	Rm				option			S	1	0	Rn				Rt						
size								opc																							

LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

```

if option<1> == '0' then UNDEFINED; // sub-word index
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale =
  UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0; 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	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	#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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, [address, datasize DIV 8, acctype] = data;

    when AccType_NORMALMemOp_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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32); [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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>], #<imm>
```

#### 64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = 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>, #<imm>]!
```

#### 64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = 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;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);(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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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    UNDEFINED;
        when Constraint_NOP      EndOfInstruction();
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(8) UNKNOWN;
data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 1, [address, datasize DIV 8, acctype] = data;
when AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	23	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	1										S	1	0									
size								opc				Rm				option				Rn				Rt							

### 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>}]

```

if option<1> == '0' then UNDEFINED; // sub-word index
boolean postindex = FALSE;
integer scale =
  UInt(size);
if option<1> == '0' then UNDEFINED; // 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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, [address, datasize DIV 8, acctype] = data;

    when AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, [address, datasize DIV 8, acctype];
        if signed then AccType_NORMAL];
        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(address, t<4:0>); [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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>], #<imm>
```

#### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = 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>, #<imm>]!
```

#### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = 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		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;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);(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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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    UNDEFINED;
        when Constraint_NOP      EndOfInstruction();
```



## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(16) UNKNOWN;
data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 2, {address, datasize DIV 8, acctype}] = data;
when AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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>}}]
```

```
if option<1> == '0' then UNDEFINED; // sub-word index
boolean postindex = FALSE;
integer scale =
  UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0; 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> 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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, [address, datasize DIV 8, acctype]] = data;

    when AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, [address, datasize DIV 8, acctype]];
        if signed then AccType_NORMAL];
        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(address, t<4:0>);[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																									

LDRSW <Xt>, [<Xn|SP>], #<sim>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = 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																																

LDRSW <Xt>, [<Xn|SP>, #<sim>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = 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																												

LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);(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);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPD);
    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 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 wback = FALSE; // writeback is suppressed
rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

if !postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 4, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMALMemOp_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(data, 64);
    (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

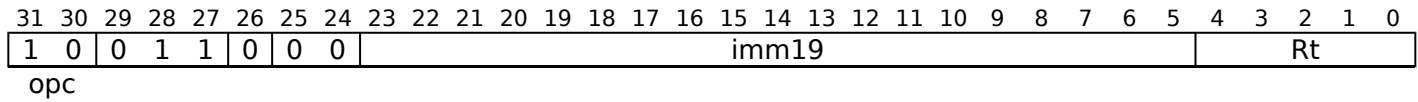
(old)

htmldiff from-

(new)

## LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



LDRSW <Xt>, <label>

```
integer t = UInt(Rt);
bits(64) offset;

offset = (Rt); MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 4;
    signed = TRUE;
  when '11'
    memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64); (imm19:'00', 64);
boolean tag_checked = FALSE;
```

## Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose 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(32) data;
bits(size*8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(FALSE);
(tag_checked);

data = case memop of
  when MemOp_LOAD
    data = Mem[address, 4, {address, size, AccType_NORMAL}];
    if signed then
X[t] = SignExtend(data, 64);
    else
      X[t] = data;

  when MemOp_PREFETCH Prefetch(data, 64); (address, t<4:0>);
```



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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	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	1	Rm				option			S	1	0	Rn				Rt						
size								opc																							

```
LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UNDEFINED; // sub-word index
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale =
  UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 2 else 0; 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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
(tag_checked);

bits(64) address;
bits(32) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 4, [address, datasize DIV 8, acctype] = data;

    when AccType_NORMALMemOp_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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 64); [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	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	0	1	1	0	0	Rn					Rt				
size											opc																				

### 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 = 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_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31; ;MemAtomicOp op;
case op 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;
boolean tag_checked = n != 31;
```

## 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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_ORR, value, ldacctype, stacctype);

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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```



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, STSETLB</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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="#">STSETHL</a> , <a href="#">STSETLH</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	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	0	0	0	0	Rn					Rt				
size											opc																				

### 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 = 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_ATOMICRW;
```

```
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

```
boolean tag_checked = n != 31; ;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;
```

```
boolean tag_checked = n != 31;
```

## 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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});
if t != 31 then MemAtomicOp_SMAX, value, ldacctype, stacctype);
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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</a> , <a href="#">STSMAXB</a>	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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(16) value;
bits(16) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	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	0	1	0	0	Rn					Rt				
size											opc																				

### 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 = 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_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31; ;MemAtomicOp op;
case op 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;
boolean tag_checked = n != 31;
```

## 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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});
if t != 31 then MemAtomicOp_SMIN, value, ldacctype, stacctype);
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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(8) value;
bits(8) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});
if t != 31 then MemAtomicOp_SMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```



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(16) value;
bits(16) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_SMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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*.

31	30	29	28	27	26	25	24	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	0	Rn				Rt									
size										opc																									

### 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 == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUA0Ext() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31; 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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype];
    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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, regsize)[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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*.

31	30	29	28	27	26	25	24	23	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	0	Rn				Rt									
size										opc																									

```
LDTRB <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset =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 == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUA0Ext() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31; 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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_STORE then
        if MemOp_PREFETCH then
            CheckSPAlignment();
            address = SP[];
        else
            address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, acctype];[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(data, 32);[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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*.

31	30	29	28	27	26	25	24	23	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	0	Rn				Rt									
size										opc																									

```
LDTRH <Wt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset =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 == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUA0Ext() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31; 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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();

```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_STORE then
        if n == 31 then
            if MemOp_PREFETCH then
                CheckSPAlignment();
                address = SP[];
            else
                address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, acctype];[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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32);[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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*.

31	30	29	28	27	26	25	24	23	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	0	Rn			Rt									
size										opc																								

### 32-bit (opc == 11)

```
LDTRSB <Wt>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (opc == 10)

```
LDTRSB <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = 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 == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUA0Ext() && 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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31); && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, acctype] = data;
        [address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, 1, acctype];
        [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>); (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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*.

31	30	29	28	27	26	25	24	23	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	0	Rn				Rt									
size										opc																									

### 32-bit (opc == 11)

```
LDTRSH <Wt>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (opc == 10)

```
LDTRSH <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = 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 == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUA0Ext() && 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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
    if size == '10' && opc<0> == '1' then UNDEFINED;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31); && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, acctype] = data;
        [address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, 2, acctype];
        [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>); [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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*.

31	30	29	28	27	26	25	24	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	0	Rn				Rt					
size									opc																						

LDTRSW <Xt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset = 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 == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUA0Ext() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31; 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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_STORE then
        if n == 31 then
            if MemOp_PREFETCH then
                CheckSPAlignment();
                address = SP[];
            else
                address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 4, acctype];[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(data, 64);[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	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 = 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_ATOMICRW;
```

```
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

```
boolean tag_checked = n != 31; MemAtomicOp op;
```

```
case op 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;
```

```
boolean tag_checked = n != 31;
```

## 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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});
if t != 31 then MemAtomicOp_UMAX, value, ldacctype, stacctype);
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## LDUMAXB, LDUMAXAB, 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, LDUMAXAB and LDUMAXALB load from memory with acquire semantics.
- LDUMAXLB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has neither acquire nor release semantics.

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

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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																																					

#### LDUMAXAB (A == 1 && R == 0)

LDUMAXAB <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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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
STUMAXB, STUMAXB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});
if t != 31 then MemAtomicOp_UMAX, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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(16) value;
bits(16) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	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	1	0	0	Rn					Rt				
size											opc																				

### 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 = 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_ATOMICRW;
```

```
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

```
boolean tag_checked = n != 31; ;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;
```

```
boolean tag_checked = n != 31;
```

## 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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});
if t != 31 then MemAtomicOp_UMIN, value, ldacctype, stacctype);
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```

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(8) value;
bits(8) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 neither acquire nor release semantics.

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 (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
```

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

```
integer s = UInt(Rs); (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_ATOMICRW;
```

```
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;
```

```
boolean tag_checked = n != 31;
```



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
STUMINH, STUMINLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, {address, op, value, ldacctype, stacctype});

if t != 31 then MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	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	0	Rn				Rt									
size										opc																									

### 32-bit (size == 10)

```
LDUR <Wt>, [<Xn|SP>{, #<simm>}]
```

### 64-bit (size == 11)

```
LDUR <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);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMALMemOp_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(data, regsize)[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

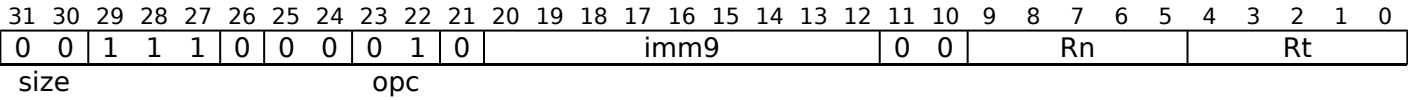
(old)

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](#).



LDURB <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset =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);

boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMALMemOp_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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32)[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

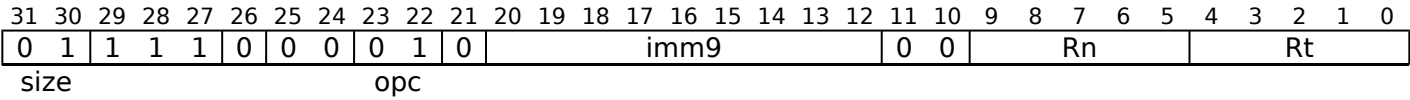
(old)

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](#).



LDURH <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset =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);

boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMALMemOp_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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 32)[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	23	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	0	Rn						Rt			
size								opc																							

### 32-bit (opc == 11)

```
LDURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

### 64-bit (opc == 10)

```
LDURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = 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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction && (n != 31);();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, [address, datasize DIV 8, acctype]] = data;

    when AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, [address, datasize DIV 8, acctype]];
        if signed then AccType_NORMAL];
        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(address, t<4:0>);[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	23	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	0	Rn						Rt			
size								opc																							

### 32-bit (opc == 11)

```
LDURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

### 64-bit (opc == 10)

```
LDURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = 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 = 32;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    // sign-extending load
    memop = if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction && (n != 31);();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, [address, datasize DIV 8, acctype]] = data;

    when AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, [address, datasize DIV 8, acctype]];
        if signed then AccType_NORMAL];
        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(address, t<4:0>); [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																											

```
LDURSW <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset =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);

boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, 4, {address, datasize DIV 8, acctype}] = data;

    when AccType_NORMALMemOp_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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X(data, 64)[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

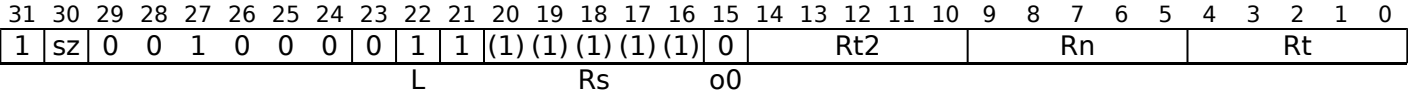
(old)

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);

integer elsize = 32 << (Rt2); // ignored by load/store single register
integer s = UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then (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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 rt_unknown = TRUE; // result is UNKNOWN
      when rn_unknown = FALSE; // address is original base
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
```

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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elseif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// 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.
when
    AArch64.SetExclusiveMonitorsMemOp_STORE(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    data = bits(datasize) UNKNOWN;
    elseif pair then
        bits(datasize DIV 2) el1 =
            X[t] = bits(datasize) UNKNOWN; // In this case t = t2
    elseif elsize == 32 then
        // 32-bit load exclusive pair (atomic)
        data = {t};
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
    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, [address, dbytes, acctype]] = data;
        status = AccType_ATOMICExclusiveMonitorsStatus;
    if(); BigEndianX([s] = AccType_ATOMICZeroExtend) then(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] = data<datasize-1:elsize>; [t] = bits(datasize) UNKNOWN; // In this case t =
        elseif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data =
                Mem[address, dbytes, acctype];
            if BigEndian(acctype) then
                X[t2] = data<elsize-1:0>;
            else [t] = data<datasize-1:elsize>;
            X[t] = data<elsize-1:0>; [t2] = data<elsize-1:0>;
        else
            X[t2] = data<datasize-1:elsize>;

```



```

else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != [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 (address, dbytes) then
                iswrite = FALSE;
                secondstage = FALSE;
                AArch64.Abort(address, AArch64.AlignmentFault((acctype, iswrite, secondstage)); AccType_ATOMICX, P
            Mem[address + 0, 8, acctype];
            X[t] = [t2] = Mem[address, 8, [address + 8, 8, acctype];
        else
            data = AccType_ATOMICMem]; [address, dbytes, acctype];
            X[t2] = [t] = MemZeroExtend[address+8, 8, AccType_ATOMIC]; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	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)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0		Rt2													

### 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 elsize = 8 << integer t2 = UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31; (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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
```

## 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(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// 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.
when
    MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
        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);

data =
    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN; // In this case t
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, {address, dbytes, acctype}];
            if (acctype) 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 = MemAccType_ATOMICBigEndian[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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													

LDXRB <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 UNDEFINED;
            when Constraint_NOP EndOfInstruction();
```

## 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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// 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.
when
    MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
        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, 1);
(address, dbytes);

data =
    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN; // In this case t
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, 1, [address, dbytes, acctype]];
            if (acctype) 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 = MemAccType_ATOMICBigEndian[address, dbytes, acctype];
    X[t] = ZeroExtend(data, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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														

LDXRH <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
```

## 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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

// 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.
when
    MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
        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, 2);
(address, dbytes);

data =
    if pair then
        if rt_unknown then
            // ConstrainedUNPREDICTABLE case X[t] = bits(datasize) UNKNOWN; // In this case t
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, 2, [address, dbytes, acctype]];
            if (acctype) 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 = MemAccType_ATOMICBigEndian[address, dbytes, acctype];
    X[t] = ZeroExtend(data, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias [MUL](#).

31	30	29	28	27	26	25	24	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	1	0	0	0	Rm				0	Ra				Rn				Rd							
																o0															

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

MADD <Wd>, <Wn>, <Wm>, <Wa>

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

MADD <Xd>, <Xn>, <Xm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32; integer destsize = if sf == '1' then 64 else 32;
integer datasize = destsize;
boolean sub_op = (o0 == '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 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.
<Wa>	Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" 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 holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">MUL</a>	Ra == '11111'

## Operation

```
bits(destsize) operand1 =bits(datasize) operand1 = X[n];
bits(destsize) operand2 =bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result =if sub_op then
result = UInt(operand3) + ((operand3) - (UInt(operand1) * UInt(operand2))); (operand2));
else
result =
UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d] = result<destsize-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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

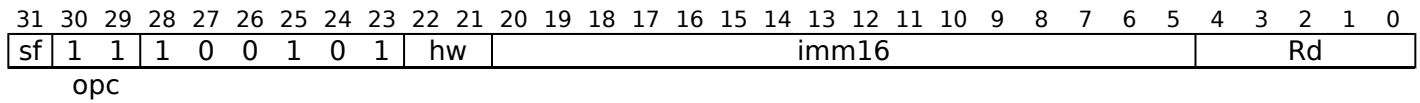
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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



## MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.



### 32-bit (sf == 0 && hw == 0x)

```
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;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = 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 = 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;

result = if opcode == MoveWideOp_K then
  result = X[d];
else
  result = Zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N[d];
result<pos+15:pos> = imm16; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

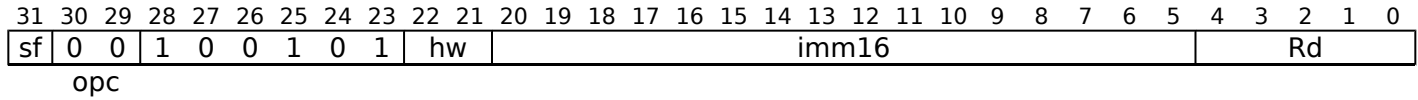
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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 && hw == 0x)

```
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;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = 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 = 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	! ( <a href="#">IsZero</a> (imm16) && hw != '00' )
<a href="#">MOV (inverted wide immediate)</a>	32-bit	! ( <a href="#">IsZero</a> (imm16) && hw != '00' ) && ! <a href="#">IsOnes</a> (imm16)

## Operation

```
bits(datasize) result;

result =if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N();

result<pos+15:pos> = imm16;
result = NOT(result);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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 \(wide immediate\)](#).

31	30	29	28	27	26	25	24	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		1		hw		imm16																Rd			
opc																																							

### 32-bit (sf == 0 && hw == 0x)

```
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;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = 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 = 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>	! ( <a href="#">IsZero</a> (imm16) && hw != '00' )

Operation

```
bits(datasize) result;

result =if opcode == MoveWideOp_K then
    result = X[d];
else
    result = Zeros();

result<pos+15:pos> = imm;
if opcode == MoveWideOp_N();

result<pos+15:pos> = imm16; 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.

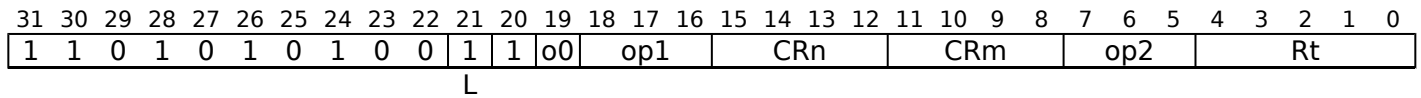
Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## MRS

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.



MRS <Xt>, (<systemreg>|S<op0>\_<op1>\_<Cn>\_<Cm>\_<op2>)

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);
```

```
integer t = UInt(Rt);
```

```
integer sys_op0 = 2 + UInt(o0);
```

```
integer sys_op1 = UInt(op1);
```

```
integer sys_op2 = UInt(op2);
```

```
integer sys_crn = UInt(CRn);
```

```
integer sys_crm = UInt(CRm);
```

```
boolean read = (L == '1');
```

## Assembler Symbols

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

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".

The System register names are defined in *'AArch64 System Registers' in the System Register XML*.

<op0> Is an unsigned immediate, encoded in "o0":

o0	<op0>
0	2
1	3

<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.

## Operation

```
if read then
```

```
    X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
```

```
else
```

```
    AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2));
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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 this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If [FEAT\\_SSBS](#) is implemented, PSTATE.SSBS.
- If [FEAT\\_PAN](#) is implemented, PSTATE.PAN.
- If [FEAT\\_UAO](#) is implemented, PSTATE.UAO.
- If [FEAT\\_DIT](#) is implemented, PSTATE.DIT.
- If [FEAT\\_MTE](#) is implemented, PSTATE.TCO.

31	30	29	28	27	26	25	24	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	



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(2) min_EL;
boolean need_secure = FALSE;
```

```
case op1 of
  when '00x'
    min_EL = EL1;
  when '010'
    min_EL = EL1;
  when '011'
    min_EL = EL0;
  when '100'
    min_EL = EL2;
  when '101'
    if !HaveVirtHostExt() then
      UNDEFINED;
    min_EL = EL2;
  when '110'
    min_EL = EL3;
  when '111'
    min_EL = EL1;
    need_secure = TRUE;
```

```
if UInt(PSTATE.EL) < UInt(min_EL) || (need_secure && !IsSecure()) then
  UNDEFINED; UNDEFINED;
```

```
bits(4) operand = CRm;
```

```
PSTATEField field;
```

```
case op1:op2 of
  when '000 011'
    if !HaveUA0Ext() then
      UNDEFINED;
    field = PSTATEField_UA0;
  when '000 100'
    if !HavePANExt() then
      UNDEFINED;
    field = PSTATEField_PAN;
  when '000 101' field = PSTATEField_SP;
  when '011 010'
    if !HaveDITExt() then
      UNDEFINED;
    field = PSTATEField_DIT;
  when '011 011'
    UNDEFINED;
  when '011 100'
    if !HaveMTEExt() then
      UNDEFINED;
    field = PSTATEField_TCO;
  when '011 110' field = PSTATEField_DAIFFSet;
  when '011 111' field = PSTATEField_DAIFFClr;
  when '011 001'
    if !HaveSSBSEExt() then
      UNDEFINED;
    field = PSTATEField_SSBS;
  otherwise UNDEFINED;
```

```
// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
```

```
if PSTATE.EL == EL0 && field IN {PSTATEField_DAIFFSet, PSTATEField_DAIFFClr} then
```

```
  if !ELUsingAArch32(EL1) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') || SCTLRL_EL1.UMA == '0') then
```

```
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
```

```
      AArch64.SystemAccessTrap(EL2, 0x18);
```

```
    else
```

```
      AArch64.SystemAccessTrap(EL1, 0x18);
```

Assembler Symbols

<pstatefield> Is a PSTATE field name, encoded in “op1:op2”:

op1	op2	<pstatefield>	Architectural Feature
000	00x	SEE_PSTATE	-
000	010	SEE_PSTATE	-
000	011	UA0	FEAT_UAO
000	100	PAN	FEAT_PAN
000	101	SPSel	-
000	11x	RESERVED	-
001	xxx	RESERVED	-
010	xxx	RESERVED	-
011	000	RESERVED	-
011	001	SSBS	FEAT_SSBS
011	010	DIT	FEAT_DIT
011	011	RESERVED	-
011	100	TC0	FEAT_MTE
011	101	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 = CRm<0>;
PSTATE.SSBS = operand<0>;
  when PSTATEField_SP
    PSTATE.SP = CRm<0>;
PSTATE.SP = operand<0>;
  when PSTATEField_DAIFSet
    PSTATE.D = PSTATE.D OR CRm<3>;
    PSTATE.A = PSTATE.A OR CRm<2>;
    PSTATE.I = PSTATE.I OR CRm<1>;
    PSTATE.F = PSTATE.F OR CRm<0>;
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(CRm<3>;);
    PSTATE.A = PSTATE.A AND NOT(CRm<2>;);
    PSTATE.I = PSTATE.I AND NOT(CRm<1>;);
    PSTATE.F = PSTATE.F AND NOT(CRm<0>;);
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 = CRm<0>;
PSTATE.PAN = operand<0>;
  when PSTATEField_UAO
    PSTATE.UAO = CRm<0>;
PSTATE.UAO = operand<0>;
  when PSTATEField_DIT
    PSTATE.DIT = CRm<0>;
PSTATE.DIT = operand<0>;
  when PSTATEField_TC0
    PSTATE.TC0 = CRm<0>;
PSTATE.TC0 = operand<0>;
```

## MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose 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	0	1	0	1	0	0	0	1	0	0	op1		CRn			CRm			op2			Rt						
											L																				

MSR (<systemreg>|S<op0>\_<op1>\_<Cn>\_<Cm>\_<op2>), <Xt>

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);
```

```
integer t = UInt(Rt);
```

```
integer sys_op0 = 2 + UInt(o0);
```

```
integer sys_op1 = UInt(op1);
```

```
integer sys_op2 = UInt(op2);
```

```
integer sys_crn = UInt(CRn);
```

```
integer sys_crm = UInt(CRm);
```

```
boolean read = (L == '1');
```

## Assembler Symbols

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".

The System register names are defined in *'AArch64 System Registers' in the System Register XML*.

<op0> Is an unsigned immediate, encoded in "o0":

o0	<op0>
0	2
1	3

<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 general-purpose source register, encoded in the "Rt" field.

## Operation

```
if read then
    X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias [MNEG](#).

31	30	29	28	27	26	25	24	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	1	0	0	0	Rm				1	Ra				Rn				Rd							
																o0															

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

MSUB <Wd>, <Wn>, <Wm>, <Wa>

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

MSUB <Xd>, <Xn>, <Xm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32; integer destsize = if sf == '1' then 64 else 32;
integer datasize = destsize;
boolean sub_op = (o0 == '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 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.
<Wa>	Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" 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 holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">MNEG</a>	Ra == '11111'

## Operation

```
bits(destsize) operand1 =bits(datasize) operand1 = X[n];
bits(destsize) operand2 =bits(datasize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result =if sub_op then
result = UInt(operand3) - (UInt(operand1) * UInt(operand2));(operand2));
else
result =
UInt(operand3) + (UInt(operand1) * UInt(operand2));
X[d] = result<destsize-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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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											

### 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 110'`

`if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP`

`op = SystemHintOp_DGH;`

`when '0000 111' SEE "XPACLRI";`

`when '0001 xxx'`

`case op2 of`

`when '000' SEE "PACIA1716";`

`when '010' SEE "PACIB1716";`

`when '100' SEE "AUTIA1716";`

`when '110' SEE "AUTIB1716";`

`otherwise EndOfInstruction(); // Instruction executes as NOP`

`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'`

`case op2 of`

`when '000' SEE "PACIAZ";`

`when '001' SEE "PACIASP";`

`when '010' SEE "PACIBZ";`

`when '011' SEE "PACIBSP";`

`when '100' SEE "AUTIAZ";`

`when '101' SEE "AUTHASP";`

`when '110' SEE "AUTIBZ";`

`when '111' SEE "AUTIBSP";`

`when '0100 xx0'`

`op = SystemHintOp_BTI;`

`// Check branch target compatibility between BTI instruction and PSTATE.BTYPE`

`SetTypeCompatible(BTypeCompatible_BTI(op2<2:1>));`

`otherwise EndOfInstruction// Empty.{}; // Instruction executes as NOP`

## Operation

```
// do nothing
case op of
  when SystemHintOp_YIELDHint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(localtimeout, WFEType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(localtimeout, WFIType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext('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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	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	1	Rm						imm6						Rn						Rd					
opc									N																								

### 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;
if sf == '0' && imm6<5> == '1' then UNDEFINED; 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;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);(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
MVN	Rn == '11111'

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);
if invert then operand2 = NOT(operand2);

result = operand1 OR 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 \(bitmask immediate\)](#).

31	30	29	28	27	26	25	24	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	0	0	N	immr						imms						Rn				Rd					
opc																															

### 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;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = 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, -) = 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;

result = operand1 OR imm;
if d == 31 then 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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;
if sf == '0' && imm6<5> == '1' then UNDEFINED; 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;

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

result = operand1 OR operand2; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																							

PRFM (<prfop>|<#imm5>), [<Xn|SP>{, #<pimm>}]

```
bits(64) offset = boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 3); (imm12, 64), scale);
```

## 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.

<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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPL);
    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 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      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    address = if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset; 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(address, t<4:0>)[n] = address;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## PRFM (literal)

Prefetch Memory (literal) 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	0	1	1	0	0	0	imm19																	Rt						

opc

PRFM (<prfop>|<#imm5>), <label>

```
integer t = UInt(Rt);
bits(64) offset;

offset = (Rt); MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 4;
    signed = TRUE;
  when '11'
    memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);(imm19:'00', 64);
boolean tag_checked = FALSE;
```

## 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>.
- <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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);(tag_checked);

case memop of
    when

MemOp_LOAD
    data = Mem[address, size, AccType_NORMAL];
    if signed then
        X[t] = SignExtend(data, 64);
    else
        X[t] = data;

    when MemOp_PREFETCHPrefetch(address, t<4:0>);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																				

PRFM (<prfop>|<#imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

```

if option<1> == '0' then UNDEFINED; // sub-word index
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale =
  UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 3 else 0; 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	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	#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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    address = if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset; 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(address, t<4:0>)[n] = address;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53 2020-12-16T14:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## PRFUM

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																									

PRFUM (<prfop>|<#imm5>), [<Xn|SP>{, #<simm>}]

```
bits(64) offset = 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.

<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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    address = if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset; 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(address, t<4:0>)[n] = address;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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

(FEAT\_SPE)

31	30	29	28	27	26	25	24	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											

PSB CSYNC

```

SystemHintOp if !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 110'
    if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_DGH;
  when '0000 111' SEE "XPACLRI";
  when '0001 xxx'
    case op2 of
      when '000' SEE "PACIA1716";
      when '010' SEE "PACIB1716";
      when '100' SEE "AUTIA1716";
      when '110' SEE "AUTIB1716";
      otherwise EndOfInstruction(); // Instruction executes as NOP
  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'
    case op2 of
      when '000' SEE "PACIAZ";
      when '001' SEE "PACIASP";
      when '010' SEE "PACIBZ";
      when '011' SEE "PACIBSP";
      when '100' SEE "AUTIAZ";
      when '101' SEE "AUTHASP";
      when '110' SEE "AUTIBZ";
      when '111' SEE "AUTIBSP";
  when '0100 xx0'
    op = SystemHintOp_BTI;
    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
    SetBTypeCompatible(BTypeCompatible_BTI() then (op2<2:1>));
  otherwise EndOfInstruction(); // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp_YIELDHint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(localtimeout, WFEType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(localtimeout, WFIType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext();('00');

  otherwise // do nothing
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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

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 && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_IND_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == BranchType_RET 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. Defaults to X30 if absent.

## Operation

```
bits(64) target = X[n];
boolean auth_then_branch = TRUE;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00'; if pac then
    bits(64) modifier = if source_is_sp then

BranchToSP(target, [] else X[m];

    if use_key_a then
        target = AuthIA(target, modifier, auth_then_branch);
    else
        target = AuthIB(target, modifier, auth_then_branch);

if branch_type == BranchType_INDCALL then
    X[30] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
case branch_type of
    when BranchType_INDIR // BR, BRAA, BRAB, BRAAZ, BRABZ
        if InGuardedPage then
            if n == 16 || n == 17 then
                BTypeNext = '01';
            else
                BTypeNext = '11';
        else
            BTypeNext = '01';
    when BranchType_INDCALL // BLR, BLRAA, BLRAB, BLRAAZ, BLRABZ
        BTypeNext = '10';
    when BranchType_RET // RET, RETAA, RETAB
        BTypeNext = '00';

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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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

(FEAT\_PAuth)

31	30	29	28	27	26	25	24	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

```

boolean use_key_a = (M == '0');

if !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 && !HavePACExt() then
    UNDEFINED;

case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_IND_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;

if pac then
    if Z == '0' && m != 31 then
        UNDEFINED;

    if branch_type == BranchType_RET() then
        UNDEFINED; then
        if n != 31 then UNDEFINED;
        n = 30;
        source_is_sp = TRUE;

```

## Operation

```
bits(64) target = X[30];
[n];
boolean auth_then_branch = TRUE;

bits(64) modifier = if pac then
    bits(64) modifier = if source_is_sp then SP[];

if use_key_a then
    target = [] else X[m];

    if use_key_a then
        target = AuthIA(target, modifier, TRUE);
else
    target = (target, modifier, auth_then_branch);
    else
        target = AuthIB(target, modifier, TRUE);
(target, modifier, auth_then_branch);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00'; if branch_type ==

BranchToBranchType_INDCALL(target, then X[30] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
case branch_type of
    when BranchType_INDIR // BR, BRAA, BRAB, BRAAZ, BRABZ
        if InGuardedPage then
            if n == 16 || n == 17 then
                BTypeNext = '01';
            else
                BTypeNext = '11';
        else
            BTypeNext = '01';
    when BranchType_INDCALL // BLR, BLRAA, BLRAB, BLRAAZ, BLRABZ
        BTypeNext = '10';
    when BranchType_RET // RET, RETAA, RETAB
        BTypeNext = '00';

BranchTo);(target, branch_type);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	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	1	x	Rn						Rd					
opc																																	

### 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;
```

## 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>;
        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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14.5341

Copyright © 2010-2020 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;
```

## 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>;
    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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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																				

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;
```

## 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>;
        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.

(old)

htmldiff from-

(new)

## 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 (FEAT\_FlagM)

31	30	29	28	27	26	25	24	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	0	0	0	imm6						0	0	0	0	1	Rn						0	mask			
sf																																

RMIF <Xn>, #<shift>, #<mask>

```
if !HaveFlagManipulateExt() then UNDEFINED;
() || sf != '1' then UNDEFINED;
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.

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias [NGC](#).

31	30	29	28	27	26	25	24	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	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
op S																															

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

SBC <Wd>, <Wn>, <Wm>

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

SBC <Xd>, <Xn>, <Xm>

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

## Alias Conditions

Alias	Is preferred when
<a href="#">NGC</a>	Rn == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcw;

operand2 = NOT(operand2);
if sub_op then
    operand2 = NOT(operand2);

(result, -) = (result, nzcw) = AddWithCarry(operand1, operand2, PSTATE.C); (operand1, operand2, PSTATE.C);
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) 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 [NGCS](#).

31	30	29	28	27	26	25	24	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	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
op S																															

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

SBCS <Wd>, <Wn>, <Wm>

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

SBCS <Xd>, <Xn>, <Xm>

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

## Alias Conditions

Alias	Is preferred when
<a href="#">NGCS</a>	Rn == '11111'

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcw;

operand2 = NOT(operand2);
if sub_op then
    operand2 = NOT(operand2);

(result, nzcw) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcw; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## SBFM

Signed 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 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 `<Wd>`, `<Wn>`, `#<immr>`, `#<imms>`

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

SBFM `<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 = 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.  
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) src =bits(datasize) dst = if inzero then Zeros() else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot =bits(datasize) bot = (dst AND NOT(wmask)) OR ( ROR(src, R) AND wmask;
(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top =bits(datasize) top = if extend then Replicate(src<S>);
(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.

31	30	29	28	27	26	25	24	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	0	0	0	1	1	Rn					Rd										
																					o1																

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

SDIV <Wd>, <Wn>, <Wm>

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

SDIV <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
boolean unsigned = (ol == '0');
```

## 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.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real((operand1, unsigned)) / Real(Int(operand2, FALSE)) / Real((operand2, unsigned)));

X[d] = result<datasize-1:0>;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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 (FEAT\_FlagM)

31	30	29	28	27	26	25	24	23	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	1	0	0	0	0	0	0	0	0	0	0	sz	0	0	1	0	Rn				0	1	1	0	1	
sf																															

sf

### SETF8 (sz == 0)

SETF8 <Wn>

### SETF16 (sz == 1)

SETF16 <Wn>

```
if !HaveFlagManipulateExt() then UNDEFINED;
integer msb = if sz == '1' then 15 else 7;
() || sf != '0' then UNDEFINED;
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';
(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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											

## 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 110'

```

```

    if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_DGH;

```

```

when '0000 111' SEE "XPACLR1";

```

```

when '0001 xxx'

```

```

    case op2 of

```

```

        when '000' SEE "PACIA1716";

```

```

        when '010' SEE "PACIB1716";

```

```

        when '100' SEE "AUTIA1716";

```

```

        when '110' SEE "AUTIB1716";

```

```

        otherwise EndOfInstruction(); // Instruction executes as NOP

```

```

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'

```

```

    case op2 of

```

```

        when '000' SEE "PACIAZ";

```

```

        when '001' SEE "PACIASP";

```

```

        when '010' SEE "PACIBZ";

```

```

        when '011' SEE "PACIBSP";

```

```

        when '100' SEE "AUTIAZ";

```

```

        when '101' SEE "AUTHASP";

```

```

        when '110' SEE "AUTIBZ";

```

```

        when '111' SEE "AUTIBSP";

```

```

when '0100 xx0'

```

```

    op = SystemHintOp_BTI;

```

```

    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE

```

```

    SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));

```

```

    otherwise EndOfInstruction// Empty.(); // Instruction executes as NOP

```

## Operation

```

case op of
  when SystemHintOp_YIELDHint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(localtimeout, WFEType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(localtimeout, WFIType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext('00');

  otherwise // do nothing

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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											

## 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 110'
    if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_DGH;
when '0000 111' SEE "XPACLRI";
when '0001 xxx'
    case op2 of
        when '000' SEE "PACIA1716";
        when '010' SEE "PACIB1716";
        when '100' SEE "AUTIA1716";
        when '110' SEE "AUTIB1716";
        otherwise EndOfInstruction(); // Instruction executes as NOP
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'
    case op2 of
        when '000' SEE "PACIAZ";
        when '001' SEE "PACIASP";
        when '010' SEE "PACIBZ";
        when '011' SEE "PACIBSP";
        when '100' SEE "AUTIAZ";
        when '101' SEE "AUTHASP";
        when '110' SEE "AUTIBZ";
        when '111' SEE "AUTIBSP";
when '0100 xx0'
    op = SystemHintOp_BTI;
    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
    SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
otherwise EndOfInstruction// Empty.(); // Instruction executes as NOP

```



## Operation

```

case op of
  when SystemHintOp_YIELDHint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(localtimeout, WFEType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(localtimeout, WFIType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext();('00');

  otherwise // do nothing

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [SMULL](#).

31	30	29	28	27	26	25	24	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	1	0	0	1	Rm				0	Ra				Rn				Rd							
U										o0																					

SMADDL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

### 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.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

### Alias Conditions

Alias	Is preferred when
<a href="#">SMULL</a>	Ra == '11111'

### Operation

```
bits(32) operand1 = bits(datasize) operand1 = X[n];
bits(32) operand2 = bits(datasize) operand2 = X[m];
bits(64) operand3 = bits(destsize) operand3 = X[a];

integer result;

result = if sub_op then
    result = Int(operand3, FALSE) + ((operand3, unsigned) - (Int(operand1, FALSE) * (operand1, unsigned)))
else
    result =
        Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```

### 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

SMC

Secure Monitor Call causes an exception to EL3.  
SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.  
If the values of *HCR\_EL2*.TSC and *SCR\_EL3*.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in *ESR\_ELx*, using the EC value 0x17, that is taken to EL3.  
If the value of *HCR\_EL2*.TSC is 1 and EL2 is enabled in the current Security state, execution of an SMC instruction at EL1 generates an exception that is taken to EL2, regardless of the value of *SCR\_EL3*.SMD. For more information, see *Traps to EL2 of Non-secure EL1 execution of SMC instructions*.  
If the value of *HCR\_EL2*.TSC is 0 and the value of *SCR\_EL3*.SMD is 1, the SMC instruction is UNDEFINED.

31	30	29	28	27	26	25	24	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	1

```
SMC #<imm>

// Empty.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

```
AArch64.CheckForSMCUndef0rTrap(imm16);(imm);
AArch64.CallSecureMonitor(imm16);(imm);
```

## SMSUBL

Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [SMNEGL](#).

31	30	29	28	27	26	25	24	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	1	0	0	1	Rm					1	Ra					Rn					Rd								
U										o0																									

SMSUBL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

### 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.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

### Alias Conditions

Alias	Is preferred when
<a href="#">SMNEGL</a>	Ra == '11111'

### Operation

```
bits(32) operand1 = bits(datasize) operand1 = X[n];
bits(32) operand2 = bits(datasize) operand2 = X[m];
bits(64) operand3 = bits(destsize) operand3 = X[a];

integer result;

result = if sub_op then
    result = Int(operand3, FALSE) - ((operand3, unsigned) - (Int(operand1, FALSE) * (operand1, unsigned)))
else
    result =
Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```

### 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit 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
1	0	0	1	1	0	1	1	0	1	0	Rm				0	(1)	(1)	(1)	(1)	(1)	Rn				Rd						
U								Ra																							

SMULH <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm); (Rm);
integer a = UInt(Ra); // ignored by UMULH/SMULH
integer destsize = 64;
integer datasize = destsize;
boolean unsigned = (U == '1');
```

### 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 holding the multiplicand, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

### Operation

```
bits(64) operand1 = bits(datasize) operand1 = X[n];
bits(64) operand2 = bits(datasize) operand2 = X[m];

integer result;

result = Int(operand1, FALSE) * (operand1, unsigned) * Int(operand2, FALSE); (operand2, unsigned);
X[d] = result<127:64>;
```

### 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST2G

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. 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 source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

### Post-index (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

ST2G <Xt|SP>, [<Xn|SP>], #<simm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;boolean postindex = TRUE;
boolean zero_data = FALSE;
```

### Pre-index (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

ST2G <Xt|SP>, [<Xn|SP>, #<simm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;boolean postindex = FALSE;
boolean zero_data = FALSE;
```

### Signed offset (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

ST2G <Xt|SP>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;boolean postindex = FALSE;
boolean zero_data = FALSE;
```

### Assembler Symbols

<Xt|SP> 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(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset; address = address + offset;

if zero_data then
    if address !=

Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, TRUE, FALSE));

    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);
    Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## ST64B

Single-copy Atomic 64-byte Store without Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location. The data that is stored is atomic and is required to be 64-byte-aligned.

### Integer (FEAT\_LS64)

31	30	29	28	27	26	25	24	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	0	0	1	1	1	1	1	1	1	0	0	1	0	0	Rn				Rt					

ST64B <Xt>, [<Xn|SP> {, #0}]

```
if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

### Assembler Symbols

<Xt> Is the 64-bit name of the first 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

```
CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

for i = 0 to 7
    value = X[t+i];
    if BigEndian(acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;
    data<63+64*i : 64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

MemStore64B(address, data, acctype);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST64BV

Single-copy Atomic 64-byte Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

### Integer

(FEAT\_LS64\_V)

31	30	29	28	27	26	25	24	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	0	0	1	Rs					1	0	1	1	0	0	Rn					Rt				

ST64BV <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
boolean tag_checked = n != 31;
```

### Assembler Symbols

- <Xs> Is the 64-bit name of the general-purpose register into which the status result of this instruction 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.
  - 0xFFFFFFFF\_FFFFFFFF** If the memory location accessed does not support this instruction.
- If XZR is used, then the return value is ignored.
- <Xt> Is the 64-bit name of the first 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

```
CheckST64BVEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

for i = 0 to 7
    value = X[t+i];
    if BigEndian(acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;
data<63+64*i : 64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

status = MemStore64BWithRet(address, data, acctype);

if s != 31 then X[s] = status;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14Z 5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ST64BV0

Single-copy Atomic 64-byte EL0 Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, with the bottom 32 bits taken from [ACCDATA\\_EL1](#), and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

### Integer

(FEAT\_LS64\_V)

31	30	29	28	27	26	25	24	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	0	0	1	Rs					1	0	1	0	0	0	Rn					Rt				

ST64BV0 <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
boolean tag_checked = n != 31;
```

### Assembler Symbols

- <Xs> Is the 64-bit name of the general-purpose register into which the status result of this instruction 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.
  - 0xFFFFFFFF\_FFFFFFFF** If the memory location accessed does not support this instruction.
- If XZR is used, then the return value is ignored.
- <Xt> Is the 64-bit name of the first 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

```
CheckST64BV0Enabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) Xt = X[t];
value<31:0> = ACCDATA_EL1<31:0>;
value<63:32> = Xt<63:32>;
if BigEndian(acctype) then value = BigEndianReverse(value);
data<63:0> = value;
for i = 1 to 7
    value = X[t+i];
    if BigEndian(acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;
data<63+64*i : 64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

status = MemStore64BWithRet(address, data, acctype);

if s != 31 then X[s] = status;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 base 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 (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

STG <Xt|SP>, [<Xn|SP>], #<sim>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;boolean postindex = TRUE;
boolean zero_data = FALSE;
```

### Pre-index (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

STG <Xt|SP>, [<Xn|SP>, #<sim>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;boolean postindex = FALSE;
boolean zero_data = FALSE;
```

### Signed offset (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

STG <Xt|SP>, [<Xn|SP>{, #<sim>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;boolean postindex = FALSE;
boolean zero_data = FALSE;
```

## Assembler Symbols

<Xt|SP>      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;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

bits(64) data = if t == 31 then if zero_data then
    if address != Align(address, TAG_GRANULE) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, TRUE, FALSE));

    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## STGM

Store Tag Multiple writes a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID\_EL1.BS, and the Allocation Tag written to address A is taken from the source register at  $4*A<7:4>+3:4*A<7:4>$ .

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

If [ID\\_AA64PFR1\\_EL1.MTE](#) != 0b0010, this instruction is UNDEFINED.

### Integer (FEAT\_MTE2)

31	30	29	28	27	26	25	24	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							

STGM <Xt>, [<Xn|SP>]

```
if !HaveMTE2Ext() then UNDEFINED;
```

```
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

```
if PSTATE.EL == EL0 then
    UNDEFINED;
```

```
bits(64) data = X[t];
bits(64) address;
```

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
```

```
integer size = 4 * (2 ^ (integer size = 4*(2^(UInt(GMID_EL1.BS))));
address = Align(address, size);
(address, size);
```

```
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);
```

```
for i = 0 to count-1
    bits(4) tag = data<(index*4)+3:index*4>;
    AArch64.MemTag[address, AccType_NORMAL] = tag;
    address = address + TAG_GRANULE;
    index = index + 1;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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 (FEAT\_MTE)

31	30	29	28	27	26	25	24	23	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						

STGP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

```
if !HaveMTEExt() then UNDEFINED;
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 (FEAT\_MTE)

31	30	29	28	27	26	25	24	23	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						

STGP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!

```
if !HaveMTEExt() then UNDEFINED;
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 (FEAT\_MTE)

31	30	29	28	27	26	25	24	23	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						

STGP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]

```
if !HaveMTEExt() then UNDEFINED;
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, a multiple of 16 in the range -1024 to 1008, encoded in the "simm7" field.
- For the signed offset variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "simm7" field.

## Operation

```

bits(64) address;
bits(64) data1;
bits(64) data2;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data1 = X[t];
data2 = X[t2];

if !postindex then
    address = address + offset;

if address != Align(address, TAG_GRANULE) then then
    iswrite = TRUE;
    secondstage = FALSE;
    AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, TRUE, FALSE));, iswrite, secondstage));

Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;

AArch64.MemTag[address, AccType_NORMAL] = AArch64.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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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 (FEAT\_LOR)

31	30	29	28	27	26	25	24	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)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size		L							Rs							o0	Rt2														

### 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 elsize = 8 << integer t2 = UInt(size);
boolean tag_checked = n != 31; (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;
boolean tag_checked = n != 31;
```

## 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(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
    Mem[address, dbytes, [address, dbytes, acctype] = data;

    when
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtendAccType_LIMITEDORDEREDMemOp_LOAD] = data; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 (FEAT\_LOR)

31	30	29	28	27	26	25	24	23	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															

STLLRB <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;
```

### 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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
Mem[address, 1, [address, dbytes, acctype]] = data;

when
    data = Mem[address, dbytes, acctype];
X[t] = ZeroExtendAccType_LIMITEDORDEREDMemOp_LOAD = data; (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 (FEAT\_LOR)

31	30	29	28	27	26	25	24	23	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															

STLLRH <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;
```

### 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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
Mem[address, 2, [address, dbytes, acctype]] = data;

when
    data = Mem[address, dbytes, acctype];
X[t] = ZeroExtendAccType_LIMITEDORDEREDMemOp_LOAD = data; (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 elsize = 8 << integer t2 = UInt(size);
boolean tag_checked = n != 31; (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;
boolean tag_checked = n != 31;
```

## 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(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
    Mem[address, dbytes, [address, dbytes, acctype] = data;

    when
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtendAccType_ORDEREDMemOp_LOAD] = data; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	23	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)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt					
size								L				Rs				o0		Rt2														

STLRB <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;
```

## 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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
Mem[address, 1, [address, dbytes, acctype] = data;

when
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtendAccType_ORDEREDMemOp_LOAD] = data; (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														

STLRH <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

```
boolean tag_checked = n != 31; 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;
boolean tag_checked = n != 31;
```

## 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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = case memop of
    when MemOp_STORE
        data = X[t];
Mem[address, 2, [address, dbytes, acctype] = data;

when
    data = Mem[address, dbytes, acctype];
    X[t] = ZeroExtendAccType_ORDEREDMemOp_LOAD] = data; (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*.

### Unscaled offset (FEAT\_LRCPC2)

31	30	29	28	27	26	25	24	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	0	0	imm9									0	0	Rn				Rt									
size										opc																									

### 32-bit (size == 10)

STLUR <Wt>, [<Xn|SP>{, #<sim>}]

### 64-bit (size == 11)

STLUR <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);

integer datasize = 8 << scale;
boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype] = data;

    when
        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
        XAccType_ORDERED[MemOp_LOAD] = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

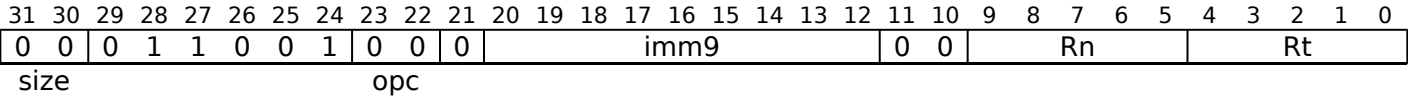
## STLURB

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*.

### Unscaled offset (FEAT\_LRCPC2)



STLURB <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset =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);

boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 1, [address, datasize DIV 8, acctype]] = data;

    when
        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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_ORDEREDMemOp_LOAD = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

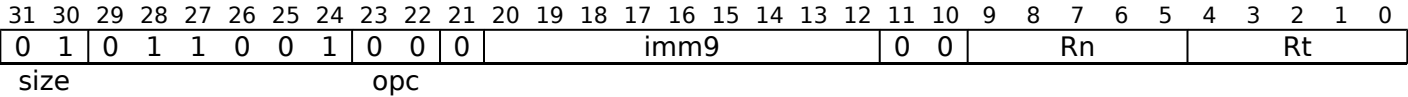
# STLURH

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*.

## Unscaled offset (FEAT\_LRCPC2)



STLURH <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset =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);

boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 2, [address, datasize DIV 8, acctype] = data;

    when
        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
        XAccType_ORDEREDMemOp_LOAD = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

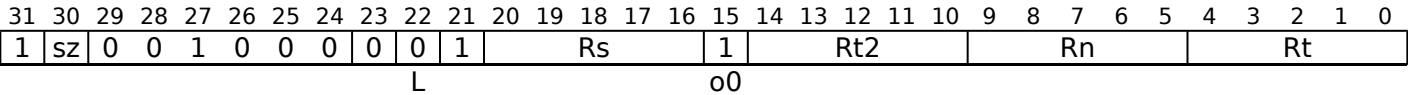
htmldiff from-

(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](#).



### 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

integer elsize = 32 << (Rs); // ignored by all loads and store-release
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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
    else
        data = AccType_ORDEREDATOMICX then el1:el2 else el2:el1;
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[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, [address, dbytes, acctype] = data;
        status = AccType_ORDEREDATOMICExclusiveMonitorsStatus = data;
        status = (); [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; // In this case t = t2
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian(acctype) 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 = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32);(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.

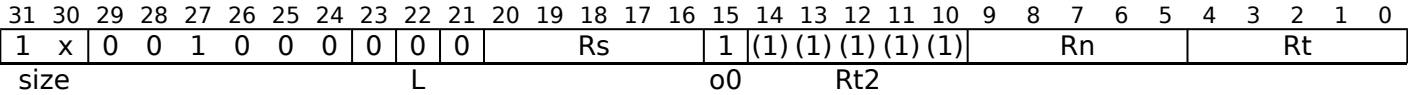
Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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*.



### 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 s = integer t2 = UInt(Rs); // ignored by all loads and store-release

integer elsize = 8 << (Rt2); // ignored by load/store single register
integer s = UInt(size);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then (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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

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(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = 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 bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
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, [address, dbytes, acctype]] = data;
        status = AccType_ORDEREDATOMICExclusiveMonitorsStatus = data;
    status = (); [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; // In this case t = t2
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian(acctype) 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 = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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															

STLXRB <Ws>, <Wt>, [<Xn|SP>{, #0}]

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then(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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();

```

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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = 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 bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
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, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, [address, dbytes, acctype] = data;
    status = AccType_ORDEREDATOMICExclusiveMonitorsStatus = data;
    status = (); [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; // In this case t = t2
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian(acctype) 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 = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## STLXRRH

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														

STLXRRH <Ws>, <Wt>, [<Xn|SP>{, #0}]

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then(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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();

```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXRRH](#).

## 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: <b>0</b> If the operation updates memory. <b>1</b> 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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = 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 bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
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, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, [address, dbytes, acctype] = data;
    status = AccType_ORDEREDATOMICExclusiveMonitorsStatus = data;
    status = (); [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; // In this case t = t2
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian(acctype) 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 = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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>}]

```
// Empty.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);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + (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 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

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 UNDEFINED;
        when Constraint_NOP EndOfInstruction(imm7, 64), scale);
boolean tag_checked = n != 31;();
```

## Operation

```

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data1 = case memop of
    when MemOp_STORE
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
    data2 = if rt_unknown && t2 == n then
        data2 = bits(datasize) UNKNOWN;
    else
        data2 = X[t2];
Mem[address, dbytes, [address + 0, dbytes, acctype] = data1; AccType_STREAMMem = data1; [address + dbytes] = data2;

    when
MemOp_LOAD
    data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype];
    data2 = [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
        XAccType_STREAMMem = data2; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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		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> 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.

## Shared Decode

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
integer scale = 2 + (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 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then if memop ==
  MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
  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 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 Constraint_NOP EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP} rt_unknown = FALSE; // value
};
case c of
  when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
rt_unknown = TRUE; // result is UNKNOWN
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP EndOfInstruction();

```

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    if ! postindex then
        address = address + offset;

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = 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, dbytes, [address + 0, dbytes, acctype] = data1; AccType_NORMALMem] = data1; [address + dbytes, dbytes, acctype] = data2;
    when
    MemOp_LOAD
        data1 = Mem[address+address, dbytes, [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;
            XAccType_NORMALMem = data2;
    [t2] = data2;

if wback then
    if postindex 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+00:00

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## 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		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.
<sim>	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);

integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPD);
    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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```



## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
    Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype] = data;

    when
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCHPrefetchAccType_NORMALMemOp_LOAD] = data;
    (address, t<4:0>);

if wback then
    if postindex then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elseif 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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	1	Rm				option		S	1	0	Rn				Rt											
size										opc																									

### 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 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);

integer datasize = 8 << scale; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then if n == 31 then
    if memop !=
        MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype] = data;

    when
        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
        XAccType_NORMAL MemOp_LOAD = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																						

STRB <Wt>, [<Xn|SP>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = 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																															

STRB <Wt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = 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																												

STRB <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);(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.
<simm>	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);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPD);
    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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
    Mem[address, 1, [address, datasize DIV 8, acctype]] = data;

    when
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetchAccType_NORMALMemOp_LOAD] = data;
(address, t<4:0>);

if wback then
    if postindex 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	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	1	Rm				option			S	1	0	Rn				Rt										
size										opc																									

### Extended register (option != 011)

STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]

### Shifted register (option == 011)

STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]

```
if option<1> == '0' then UNDEFINED; // sub-word index
boolean postindex = FALSE;
integer scale =
  UInt(size);
if option<1> == '0' then UNDEFINED; // 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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then if n == 31 then
    if memop !=
        MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 1, [address, datasize DIV 8, acctype] = data;

    when
        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
        XAccType_NORMALMemOp_LOAD] = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																									

STRH <Wt>, [<Xn|SP>], #<sim>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = 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																																

STRH <Wt>, [<Xn|SP>, #<sim>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = 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	1	0	0		1	0		0		imm12											Rn				Rt					
size										opc																									

STRH <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);(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);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPD);
    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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            else
                data = X[t];
    Mem[address, 2, [address, datasize DIV 8, acctype]] = data;

    when
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCHPrefetchAccType_NORMALMemOp_LOAD] = data;
    (address, t<4:0>);

if wback then
    if postindex 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	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	1	Rm				option		S	1	0	Rn				Rt											
size											opc																								

STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

```

if option<1> == '0' then UNDEFINED; // sub-word index
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale =
  UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0; 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	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	#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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();

```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 2, [address, datasize DIV 8, acctype] = data;

    when
        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
        XAccType_NORMALMemOp_LOAD = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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*.

31	30	29	28	27	26	25	24	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	0	Rn				Rt									
size										opc																									

### 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 == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUA0Ext() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

integer datasize = 8 << scale;
boolean tag_checked = n != 31; 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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = 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[address, datasize DIV 8, acctype] = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

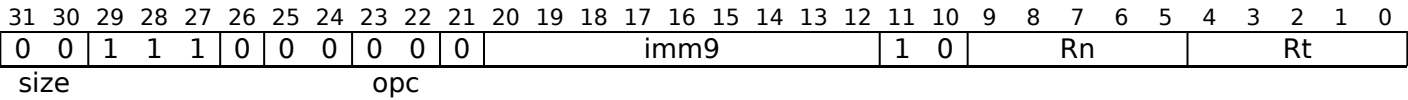
STTRB

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*.



STTRB <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset =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 == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUA0Ext() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31; 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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = 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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[address, 1, acctype] = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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*.

31	30	29	28	27	26	25	24	23	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	0	Rn				Rt									
size										opc																									

STTRH <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset = 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 == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUA0Ext() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31; 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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```



## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = 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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[address, 2, acctype] = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	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	0	Rn				Rt									
size										opc																									

### 32-bit (size == 10)

STUR <Wt>, [<Xn|SP>{, #<simm>}]

### 64-bit (size == 11)

STUR <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);

integer datasize = 8 << scale;
boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype] = data;

    when
        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
        XAccType_NORMALMemOp_LOAD = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

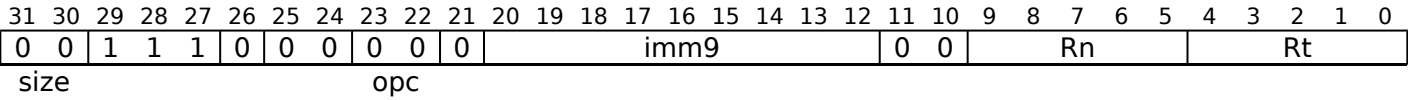
(old)

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](#).



STURB <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset =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);

boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 1, [address, datasize DIV 8, acctype] = data;

    when
        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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_NORMALMemOp_LOAD = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

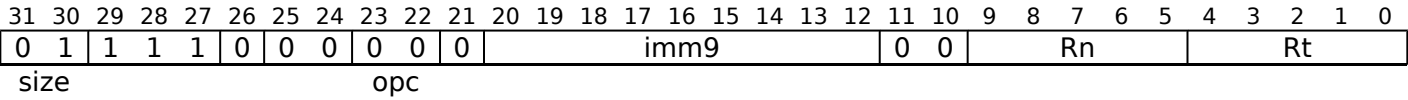
(old)

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](#).



STURH <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset =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);

boolean tag_checked = n != 31; (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 = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
bits(datasize) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
Mem[address, 2, [address, datasize DIV 8, acctype] = data;

    when
        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;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        XAccType_NORMALMemOp_LOAD = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

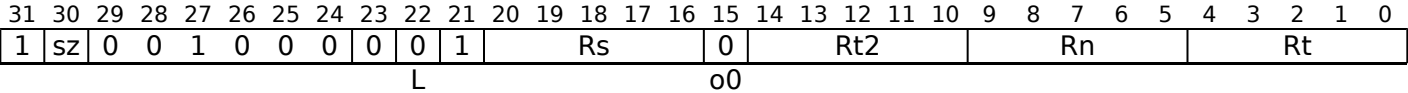
(old)

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

integer elsize = 32 << (Rs); // ignored by all loads and store-release
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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
        bits(datasize DIV 2) el2 = X[t2];
        data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
    else
        data = AccType_ATOMICX then el1:el2 else el2:el1;
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[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, [address, dbytes, acctype] = data;
        status = AccType_ATOMICExclusiveMonitorsStatus = data;
        status = (); [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; // In this case t = t2
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian(acctype) 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 = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

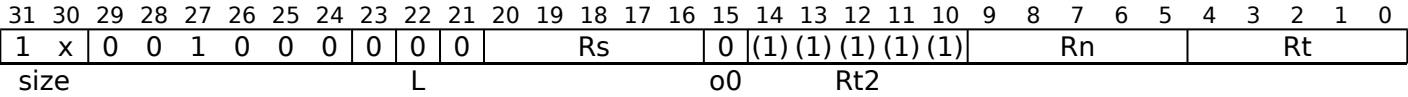
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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](#).



### 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 s = integer t2 = UInt(Rs); // ignored by all loads and store-release

integer elsize = 8 << (Rt2); // ignored by load/store single register
integer s = UInt(size);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then (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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

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(elsize) data;
constant integer dbytes = elsize DIV 8;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = 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 bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
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, [address, dbytes, acctype]] = data;
        status = AccType_ATOMICExclusiveMonitorsStatus = data;
    status = (); [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; // In this case t = t2
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian(acctype) 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 = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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)										
size								L				o0				Rt2															

STXRB <Ws>, <Wt>, [<Xn|SP>{, #0}]

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then (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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();

```

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(8) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = 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 bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
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, 1) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, 1, [address, dbytes, acctype] = data;
        status = AccType_ATOMICEExclusiveMonitorsStatus = data;
        status = (); [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; // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian(acctype) 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 = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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)										Rt
size								L				o0				Rt2															

STXRH <Ws>, <Wt>, [<Xn|SP>{, #0}]

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = integer t2 = UInt(Rs); // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then (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;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if memop == 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 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 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 UNDEFINED;
        when Constraint_NOP EndOfInstruction();

```

## 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: <b>0</b> If the operation updates memory. <b>1</b> 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(16) data;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(datasize) UNKNOWN;
            elsif pair then
                bits(datasize DIV 2) el1 = 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 bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian(acctype) then el1 : el2 else el2 : el1;
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, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, [address, dbytes, acctype] = data;
    status = AccType_ATOMICEExclusiveMonitorsStatus = data;
    status = (); [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; // In this case t = t2
        elsif elsize == 32 then
            // 32-bit load exclusive pair (atomic)
            data = Mem[address, dbytes, acctype];
            if BigEndian(acctype) 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 = MemExclusiveMonitorsStatusX();[address, dbytes, acctype];
X[s] = [t] = ZeroExtend(status, 32);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 base 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 (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

STZ2G <Xt|SP>, [<Xn|SP>], #<sim>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE; boolean postindex = TRUE;
boolean zero_data = TRUE;
```

### Pre-index (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

STZ2G <Xt|SP>, [<Xn|SP>, #<sim>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE; boolean postindex = FALSE;
boolean zero_data = TRUE;
```

### Signed offset (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

STZ2G <Xt|SP>, [<Xn|SP>{, #<sim>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE; boolean postindex = FALSE;
boolean zero_data = TRUE;
```

## Assembler Symbols

<Xt|SP>      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(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if address != if zero_data then
    if address != Align(address, TAG_GRANULE) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);
Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

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 base 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 (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

STZG <Xt|SP>, [<Xn|SP>], #<simm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE; boolean postindex = TRUE;
boolean zero_data = TRUE;
```

### Pre-index (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

STZG <Xt|SP>, [<Xn|SP>, #<simm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE; boolean postindex = FALSE;
boolean zero_data = TRUE;
```

### Signed offset (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				Xt					

STZG <Xt|SP>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE; boolean postindex = FALSE;
boolean zero_data = TRUE;
```

## Assembler Symbols

<Xt|SP>      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;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if address != if zero_data then
    if address != Align(address, TAG_GRANULE) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## STZGM

Store Tag and Zero Multiple writes a naturally aligned block of N Allocation Tags and stores zero to the associated data locations, where the size of N is identified in DCZID\_EL0.BS, and the Allocation Tag written to address A is taken from the source register bits<3:0>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

If `ID_AA64PFR1_EL1.MTE` != 0b0010, this instruction is UNDEFINED.

### Integer (FEAT\_MTE2)

31	30	29	28	27	26	25	24	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	0	0	0	0	0	0	0	0	0	0	0	Xn						Xt					

STZGM <Xt>, [<Xn|SP>]

```
if !HaveMTE2Ext() then UNDEFINED;
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

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t];
bits(4) tag = data<3:0>;
bits(64) address;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

integer size = 4 * (2 ^ (integer size = 4*(2^(UInt(DCZID_EL0.BS))));
address = Align(address, size);
(address, size);
integer count = size >> LOG2_TAG_GRANULE;

for i = 0 to count-1
    AArch64.MemTag[address, AccType_NORMAL] = tag;
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8*(8* TAG_GRANULE));
    address = address + TAG_GRANULE;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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; 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;
```

## 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) nzcvc;
bit carry_in;

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

(result, nzcvc) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

if d == 31 then 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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	0	1	0	sh	imm12												Rn						Rd			
op S																															

### 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 sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:(imm12 : Zeros(12)), datasize);
```

## 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 "sh":

sh	<shift>
0	LSL #0
1	LSL #12

## Operation

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

operand2 = NOT(imm);
(result, -) = if sub_op then
  operand2 = NOT(operand2);
  carry_in = '1';
else
  carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

if d == 31 then 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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 \(shifted register\)](#).

31	30	29	28	27	26	25	24	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	shift	0	Rm						imm6						Rn						Rd			
op S																															

### 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;

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
NEG (shifted register)	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;

operand2 = NOT(operand2);
(result, -) =if_sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';

(result, nzcvc) = AddWithCarry(operand1, operand2, '1'); (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

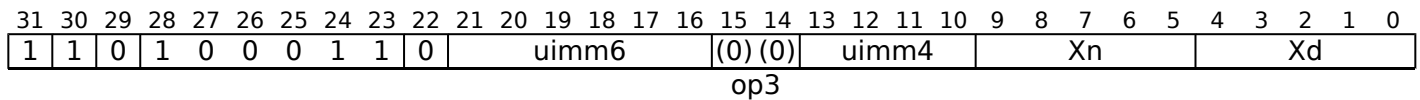
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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 (FEAT\_MTE)



SUBG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>

```
if !HaveMTEExt() then UNDEFINED;
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 = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
    (start_tag, tag_offset, exclude);
else
    rtag = '0000';

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

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

result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, rtag);

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

(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 (FEAT\_MTE)

31	30	29	28	27	26	25	24	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				

SUBP <Xd>, <Xn|SP>, <Xm|SP>

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm); (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, -) = (result, nzcvc) = AddWithCarry(operand1, operand2, '1'); (operand1, operand2, '1');

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

X[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 (FEAT\_MTE)

31	30	29	28	27	26	25	24	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						

SUBPS <Xd>, <Xn|SP>, <Xm|SP>

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm); {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');

PSTATE.<N,Z,C,V> = nzcvc; if setflags then
PSTATE.<N,Z,C,V> = nzcvc;
X[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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; 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;
```

## 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="#">CMP (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;

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

(result, nzcvc) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcvc; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(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 \(immediate\)](#).

31	30	29	28	27	26	25	24	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	0	1	0	sh	imm12												Rn						Rd			
op S																															

### 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 sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:(imm12-Zeros(12)), datasize);
```

## 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 "sh":

sh	<shift>
0	LSL #0
1	LSL #12

## 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;
bits(datasize) operand2 = imm;
bits(4) nzcvc;
bit carry_in;

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

(result, nzcvc) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcvc; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	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	shift	0	Rm						imm6						Rn						Rd			
op S																															

### 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;

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' && Rd != '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;

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

(result, nzcvc) = AddWithCarry(operand1, operand2, '1');
(operand1, operand2, carry_in);

PSTATE.<N,Z,C,V> = nzcvc;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.

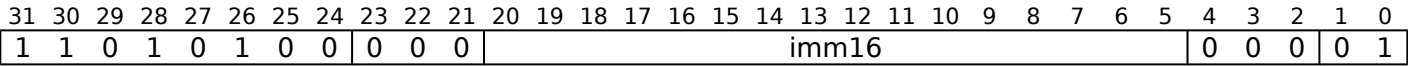
Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:20Z 2020-12-16T14:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

# SVC

Supervisor Call causes an exception to be taken to EL1.  
On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in *ESR\_ELx*, using the EC value 0x15, and the value of the immediate argument.



```
SVC #<imm>

// Empty.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

```
AArch64.CheckForSVCTrap(imm16);(imm);
AArch64.CallSupervisor(imm16);(imm);
```



## 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 neither acquire nor release semantics.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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					1		0		0		0		0		0		Rn					Rt				
size																																																

#### 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 = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); (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_ATOMICRW;
```

```
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

```
boolean tag_checked = n != 31;
```

### 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(8) data;
bits(8) store_value;
bits(datasize) data;
bits(datasize) store_value;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 neither acquire nor release semantics.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

### Integer

(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	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 = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs); (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_ATOMICRW;
```

```
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

```
boolean tag_checked = n != 31;
```

### 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(16) data;
bits(16) store_value;
bits(datasize) data;
bits(datasize) store_value;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, 32);(data, regsize);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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](#).

31	30	29	28	27	26	25	24	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			CRn			CRm			op2			Rt						
										L																					

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); {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 '11111', encoded in the "Rt" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">AT</a>	CRn == '0111' && CRm == '100x' && SysOp(op1, '0111', CRm, op2) == <a href="#">Sys_AT</a>
<a href="#">CFP</a>	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '100'
<a href="#">CPP</a>	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '111'
<a href="#">DC</a>	CRn == '0111' && SysOp(op1, '0111', CRm, op2) == <a href="#">Sys_DC</a>
<a href="#">DVP</a>	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '101'
<a href="#">IC</a>	CRn == '0111' && SysOp(op1, '0111', CRm, op2) == <a href="#">Sys_IC</a>
<a href="#">TLBI</a>	CRn == '1000' && SysOp(op1, '1000', CRm, op2) == <a href="#">Sys_TLBI</a>

## Operation

```
if has_result then
    // No architecturally defined instructions here.
    X[t] = AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
else
    AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, {sys_op0, sys_op1, sys_crn, sys_crm, sys_op2,
```

(old)

htmldiff from-

(new)

(old)

htmldiff from-

(new)

## SYSL

System instruction with result. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of 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	1	0	1	op1			CRn			CRm			op2			Rt						
										L																					

SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>

```
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); {CRm};
```

```
boolean has_result = (L == '1');
```

## Assembler Symbols

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

## Operation

```
// No architecturally defined instructions here. if has_result then
```

```
// No architecturally defined instructions here.
```

```
X[t] = AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys_crm, sys_op2); {sys_op0, sys_op1, sys_crn, sys_op2};
```

```
else AArch64.SysInstr(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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	1	b40						imm14													Rt				
op																															

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> == op thenif operand<bit_pos> == bit_val then
    BranchTo(PC[] + offset, BranchType_DIR);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## 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.

31	30	29	28	27	26	25	24	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	0	b40						imm14														Rt			
op																															

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> == op thenif operand<bit_pos> == bit_val then
    BranchTo(PC[] + offset, BranchType_DIR);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If [FEAT\\_TRF](#) is not implemented, this instruction executes as a NOP.

### System (FEAT\_TRF)

31	30	29	28	27	26	25	24	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											

TSB CSYNC

```

SystemHintOp if !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 110'
        if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_DGH;
    when '0000 111' SEE "XPACLR1";
    when '0001 xxx'
        case op2 of
            when '000' SEE "PACIA1716";
            when '010' SEE "PACIB1716";
            when '100' SEE "AUTIA1716";
            when '110' SEE "AUTIB1716";
            otherwise EndOfInstruction(); // Instruction executes as NOP
    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'
        case op2 of
            when '000' SEE "PACIAZ";
            when '001' SEE "PACIASP";
            when '010' SEE "PACIBZ";
            when '011' SEE "PACIBSP";
            when '100' SEE "AUTIAZ";
            when '101' SEE "AUTHASP";
            when '110' SEE "AUTIBZ";
            when '111' SEE "AUTIBSP";
    when '0100 xx0'
        op = SystemHintOp_BTI;
        // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
        SetBTypeCompatible(BTypeCompatible_BTI() then (op2<2:1>));
    otherwise EndOfInstruction(); // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp_YIELDHint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(localtimeout, WFEType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(localtimeout, WFIType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext();('00');

  otherwise // do nothing
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UBFM

Unsigned 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 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 `<Wd>`, `<Wn>`, `#<immr>`, `#<imms>`

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

UBFM `<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 = UInt(immr);
(wmask, tmask) = S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

## 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;immr&gt;</code>	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.
<code>&lt;imms&gt;</code>	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) src = bits(datasize) dst = if inzero then Zeros() else X[n];
// perform bitfield move on low bits
bits(datasize) bot = [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, R) AND wmask;
(src<S>) else dst;

// combine extension bits and result bits
X[d] = bot AND tmask; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

31	30	29	28	27	26	25	24	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	0	0	0	1	0	Rn					Rd										
																					o1																

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

UDIV <Wd>, <Wn>, <Wm>

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

UDIV <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32; integer datasize = if sf == '1' then 64 else 32;
boolean unsigned = (o1 == '0');
```

## 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.

## Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real((operand1, unsigned)) / Real(Int(operand2, TRUE)))

X[d] = result<datasize-1:0>;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [UMULL](#).

31	30	29	28	27	26	25	24	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	1	1	0	1	Rm				0	Ra				Rn				Rd							
U																o0															

UMADDL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

### 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.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

### Alias Conditions

Alias	Is preferred when
<a href="#">UMULL</a>	Ra == '11111'

### Operation

```
bits(32) operand1 =bits(datasize) operand1 = X[n];
bits(32) operand2 =bits(datasize) operand2 = X[m];
bits(64) operand3 =bits(destsize) operand3 = X[a];

integer result;

result =if sub_op then
    result = Int(operand3, TRUE) + ((operand3, unsigned) - (Int(operand1, TRUE) *(operand1, unsigned) *
else
    result =

Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```

### 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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



## UMSUBL

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [UMNEGL](#).

31	30	29	28	27	26	25	24	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	1	1	0	1	Rm				1	Ra				Rn				Rd							
U										o0																					

UMSUBL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = 64;
integer datasize = 32;
boolean sub_op = (o0 == '1');
boolean unsigned = (U == '1');
```

### 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.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

### Alias Conditions

Alias	Is preferred when
<a href="#">UMNEGL</a>	Ra == '11111'

### Operation

```
bits(32) operand1 = bits(datasize) operand1 = X[n];
bits(32) operand2 = bits(datasize) operand2 = X[m];
bits(64) operand3 = bits(destsize) operand3 = X[a];

integer result;

result = if sub_op then
    result = Int(operand3, TRUE) - ((operand3, unsigned) - (Int(operand1, TRUE) * (operand1, unsigned) *
else
    result =
Int(operand3, unsigned) + (Int(operand1, unsigned) * Int(operand2, unsigned));

X[d] = result<63:0>;
```

### 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit 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
1	0	0	1	1	0	1	1	1	1	0	Rm					0	(1)	(1)	(1)	(1)	(1)	Rn					Rd				
U											Ra																				

UMULH <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm); (Rm);
integer a = UInt(Ra); // ignored by UMULH/SMULH
integer destsize = 64;
integer datasize = destsize;
boolean unsigned = (U == '1');
```

### 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 holding the multiplicand, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

### Operation

```
bits(64) operand1 = bits(datasize) operand1 = X[n];
bits(64) operand2 = bits(datasize) operand2 = X[m];

integer result;

result = Int(operand1, TRUE) * (operand1, unsigned) * Int(operand2, TRUE); (operand2, unsigned);
X[d] = result<127:64>;
```

### 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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															

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 110'
        if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_DGH;
    when '0000 111' SEE "XPACLR1";
    when '0001 xxx'
        case op2 of
            when '000' SEE "PACIA1716";
            when '010' SEE "PACIB1716";
            when '100' SEE "AUTIA1716";
            when '110' SEE "AUTIB1716";
            otherwise EndOfInstruction(); // Instruction executes as NOP
    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'
        case op2 of
            when '000' SEE "PACIAZ";
            when '001' SEE "PACIASP";
            when '010' SEE "PACIBZ";
            when '011' SEE "PACIBSP";
            when '100' SEE "AUTIAZ";
            when '101' SEE "AUTHASP";
            when '110' SEE "AUTIBZ";
            when '111' SEE "AUTIBSP";
    when '0100 xx0'
        op = SystemHintOp_BTI;
        // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
        SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
    otherwise EndOfInstruction// Empty.(); // Instruction executes as NOP

```

## Operation

```

case op of
  when SystemHintOp_YIELDHint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(-1, (localtimeout, WFXType_WFE));

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(localtimeout, WFXType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext); ('00');

otherwise // do nothing

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## WFET

Wait For Event with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or 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 WFET 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](#).

## System

(FEAT\_WFXT)

31	30	29	28	27	26	25	24	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	0	1	0	0	0	0	0	0	0					Rd
																								op2							

WFET <Xt>

```
if !HaveFeatWFXT() then UNDEFINED;

integer d = UInt(Rd); SystemHintOp op;

case op2 of
  when '000' op = SystemHintOp_WFET;
  when '001' op = SystemHintOp_WFIT;
  otherwise // Do nothing
```

## Assembler Symbols

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

## Operation

```
bits(64) operand = X[d];
integer localtimeout = UInt(operand);

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFXTDEBUG) then
  EndOfInstruction();();

case op of
  when
    SystemHintOp_WFETHint_WFE(localtimeout, WFXType_WFET);
    when SystemHintOp_WFETHint_WFI(localtimeout, WFXType_WFIT);
  otherwise
    // Instruction executes as NOP
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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															

### 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 110'
        if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
        op = SystemHintOp_DGH;
    when '0000 111' SEE "XPACLRI";
    when '0001 xxx'
        case op2 of
            when '000' SEE "PACIA1716";
            when '010' SEE "PACIB1716";
            when '100' SEE "AUTIA1716";
            when '110' SEE "AUTIB1716";
            otherwise EndOfInstruction(); // Instruction executes as NOP
    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'
        case op2 of
            when '000' SEE "PACIAZ";
            when '001' SEE "PACIASP";
            when '010' SEE "PACIBZ";
            when '011' SEE "PACIBSP";
            when '100' SEE "AUTIAZ";
            when '101' SEE "AUTHASP";
            when '110' SEE "AUTIBZ";
            when '111' SEE "AUTIBSP";
    when '0100 xx0'
        op = SystemHintOp_BTI;
        // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
        SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
    otherwise EndOfInstruction// Empty.{}; // Instruction executes as NOP

```

## Operation

```
case op of
  when SystemHintOp_YIELDHint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(localtimeout, WFXType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(-1, (localtimeout, WFXType_WFI));

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext); ('00');

otherwise // do nothing
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## WFIT

Wait For Interrupt with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. For more information, see [Wait For Interrupt](#).

As described in [Wait For Interrupt](#), the execution of a WFIT 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.](#)

### System

(FEAT\_WFXT)

31	30	29	28	27	26	25	24	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	0	1	0	0	0	0	0	0	0	1				Rd
																														op2	

WFIT <Xt>

```
if !HaveFeatWFXT() then UNDEFINED;

integer d = UInt(Rd); SystemHintOp op;

case op2 of
  when '000' op = SystemHintOp_WFET;
  when '001' op = SystemHintOp_WFIT;
  otherwise // Do nothing
```

### Assembler Symbols

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

### Operation

```
bits(64) operand = X[d];
integer localtimeout = UInt(operand);

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFXTDEBUG) then
  EndOfInstruction();();

case op of
  when
    SystemHintOp_WFETHint_WFE(localtimeout, WFXType_WFET);
  when SystemHintOp_WFETHint_WFI(localtimeout, WFXType_WFIT);
  otherwise
    // Instruction executes as NOP
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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

(FEAT\_PAAuth)

31	30	29	28	27	26	25	24	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	1	0	0	0	D	1	1	1	1	1	Rd					
Rn																															

### XPACD (D == 1)

XPACD <Xd>

### XPACI (D == 0)

XPACI <Xd>

```
boolean data = (D == '1');
integer d = UInt(Rd);
if !integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if n != 31 then UNDEFINED;
```

### System

(FEAT\_PAAuth)

31	30	29	28	27	26	25	24	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	1	1	1	1	1	1	1

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);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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											

### 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 110'
    if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_DGH;
  when '0000 111' SEE "XPACLRI";
  when '0001 xxx'
    case op2 of
      when '000' SEE "PACIA1716";
      when '010' SEE "PACIB1716";
      when '100' SEE "AUTIA1716";
      when '110' SEE "AUTIB1716";
      otherwise EndOfInstruction(); // Instruction executes as NOP
  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'
    case op2 of
      when '000' SEE "PACIAZ";
      when '001' SEE "PACIASP";
      when '010' SEE "PACIBZ";
      when '011' SEE "PACIBSP";
      when '100' SEE "AUTIAZ";
      when '101' SEE "AUTHASP";
      when '110' SEE "AUTIBZ";
      when '111' SEE "AUTIBSP";
  when '0100 xx0'
    op = SystemHintOp_BTI;
    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
    SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
  otherwise EndOfInstruction// Empty.{}; // Instruction executes as NOP

```

## Operation

```

case op of
  when SystemHintOp_YIELD Hint_Yield();

  when SystemHintOp_DGHHint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFE(localtimeout, WFEType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = -1; // No local timeout event is generated
    Hint_WFI(localtimeout, WFIType_WFI);

  when SystemHintOp_SEVSendEvent();

  when SystemHintOp_SEVLSendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESB0operation();
    TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSBProfilingSynchronizationBarrier();

  when SystemHintOp_TSBTraceSynchronizationBarrier();

  when SystemHintOp_CSDBConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_BTISetBTypeNext();('00');

  otherwise // do nothing

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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.

BFCVT: Floating-point convert from single-precision to BFloat16 format (scalar).

BFCVTN, BFCVTN2: Floating-point convert from single-precision to BFloat16 format (vector).

[BFDOT \(by element\)](#): BFloat16 floating-point dot product (vector, by element).

[BFDOT \(vector\)](#): BFloat16 floating-point dot product (vector).

[BFMLALB, BFMLALT \(by element\)](#): BFloat16 floating-point widening multiply-add long (by element).

[BFMLALB, BFMLALT \(vector\)](#): BFloat16 floating-point widening multiply-add long (vector).

BFMMLA: BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix.

[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.

[RAX1](#): 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).

[SMMLA \(vector\)](#): Signed 8-bit integer matrix multiply-accumulate (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.

SQRDMAH (by element): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

SQRDMAH (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.

SQXTN, SQXTN2: Signed saturating extract Narrow.

SQXTUN, SQXTUN2: 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).

SSHL: 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.

[SUDOT \(by element\)](#): Dot product with signed and unsigned integers (vector, by element).

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).

[UMMLA \(vector\)](#): Unsigned 8-bit integer matrix multiply-accumulate (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).

[USDOT \(by element\)](#): Dot Product with unsigned and signed integers (vector, by element).

[USDOT \(vector\)](#): Dot Product with unsigned and signed integers (vector).

USHL: Unsigned Shift Left (register).

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

USHR: Unsigned Shift Right (immediate).

[USMMLA \(vector\)](#): Unsigned and signed 8-bit integer matrix multiply-accumulate (vector).

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).

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 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				

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

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

if size != '11' then UNDEFINED;

integer esize = 8 << UInt(size);
integer datasize = esize * 2; 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}; ReduceOp_ADD, 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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				

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 << UInt(size);
integer datasize = if Q == '1' then 128 else 64; 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}; ReduceOp_ADD, 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)

## 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																															

AESD <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAEEExt() then UNDEFINED;() then UNDEFINED;
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;
result =if decrypt then
    AESInvSubBytes(AESInvShiftRows(result));(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## AESE

AESE 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																															

AESE <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAEEExt() then UNDEFINED;() then UNDEFINED;
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;
result =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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																															

AESIMC <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAEEExt() then UNDEFINED;() then UNDEFINED;
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;
result =if decrypt then
    result = AESInvMixColumns(operand);{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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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															

AESMC <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAEEExt() then UNDEFINED;() then UNDEFINED;
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;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## AND (vector)

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD&FP registers, 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.

31	30	29	28	27	26	25	24	23	22	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	Rm					0	0	0	1	1	1	Rn					Rd				
size																															

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

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

```
boolean invert = (size<0> == '1'); LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

## 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.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = operand1 AND operand2; if invert then operand2 = NOT(operand2);

case op of
  when
    LogicalOp_AND
      result = operand1 AND operand2;
  when LogicalOp_ORR
      result = operand1 OR operand2;

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)

## BFDOT (by element)

BFloat16 floating-point dot product (vector, by element). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Each pair of elements in the first source vector is multiplied by the specified pair of elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element of the destination vector that aligns with the pair of BF16 values in the first source vector. The instruction ignores the *FPCR* and does not update the *FPSR* exception status.

The BF16 pair within the second source vector is specified using an immediate index. The index range is from 0 to 3 inclusive. *ID\_AA64ISAR1\_EL1*.BF16 indicates whether this instruction is supported.

### Vector (FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	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	1	L	M	Rm				1	1	1	1	H	0	Rn				Rd					

BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm> .2H[<index>]

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

### 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	4H
1	8H

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

<index> Is the immediate index of a pair of 16-bit elements in the range 0 to 3, encoded in the "H:L" fields.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2*i+0, 16];
[operand2, 2 * i + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2*i+1, 16];
[operand2, 2 * i + 1, 16];

    bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
    Elem[result, e, 32] = BFAdd(Elem[operand3, e, 32], sum);

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## BFDOT (vector)

BFloat16 floating-point dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element of the destination vector that aligns with the pair of BF16 values in the first source vector. The instruction ignores the *FPCR* and does not update the *FPSR* exception status.

### Vector (FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	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				1	1	1	1	1	1	Rn				Rd						

```
BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

### 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	4H
1	8H
- <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;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2*e+0, 16];
[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2*e+1, 16];
[operand2, 2 * e + 1, 16];

    bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
    Elem[result, e, 32] = BFAdd(Elem[operand3, e, 32], sum);

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## BFMLALB, BFMLALT (by element)

BFloat16 floating-point widening multiply-add long (by element) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and the indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

This performs a fused multiply-add without intermediate rounding that honors all of the control bits in the *FPCR* that apply to single-precision arithmetic, including the rounding mode. It can also generate a floating-point exception that causes cumulative exception bits in the *FPSR* to be set, or a synchronous exception to be taken, depending on the enable bits in the *FPCR*. *ID\_AA64ISAR1\_EL1*.BF16 indicates whether this instruction is supported.

### Vector (FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	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	1	L	M		Rm			1	1	1	1	H	0										Rd

BFMLAL<bt> <Vd>.4S, <Vn>.8H, <Vm>.H[<index>]

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt('0':Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

### Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in "Q":

Q	<bt>
0	B
1	T

<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, in the range V0 to V15, encoded in the "Rm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) result;
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) operand3 = V[d];
bits(32) element2 = Elem[operand2, index, 16]:[operand2, index, 16]:Zeros(16);

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2*e+sel, 16]:[operand1, 2 * e + sel, 16]:Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(addend, element1, element2, FPCR[]);

V[d] = result;
```

(old)

htmldiff from-

(new)

## BFMLALB, BFMLALT (vector)

BFloat16 floating-point widening multiply-add long (vector) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

This performs a fused multiply-add without intermediate rounding that honors all of the control bits in the *FPCR* that apply to single-precision arithmetic, including the rounding mode. It can also generate a floating-point exception that causes cumulative exception bits in the *FPSR* to be set, or a synchronous exception to be taken, depending on the enable bits in the *FPCR*. *ID\_AA64ISAR1\_EL1*.BF16 indicates whether these instruction is supported.

### Vector (FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	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					1	1	1	1	1	1	Rn					Rd				

BFMLAL<bt> <Vd>.4S, <Vn>.8H, <Vm>.8H

```
if !HaveBF16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

### Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in "Q":

Q	<bt>
0	B
1	T

<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

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) operand3 = V[d];
bits(128) result;

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2*e+sel, 16]:[operand1, 2 * e + sel, 16] :Zeros(16);
    bits(32) element2 = Elem[operand2, 2*e+sel, 16]:[operand2, 2 * e + sel, 16] :Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(addend, element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## 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 '0xx01' operation = when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx11' operation = when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x01' operation = when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x1' operation = when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '1110x' operation = when '11110' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    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>”:

<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>”:

<b>cmode&lt;2:1&gt;</b>	<b>&lt;amount&gt;</b>
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

**(new)**

## BIC (vector, register)

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, 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.

31	30	29	28	27	26	25	24	23	22	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	Rm					0	0	0	1	1	1	Rn					Rd				
size																															

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

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

boolean invert = (size<0> == '1'); LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

## 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.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

operand2 = NOT(operand2);
if invert then operand2 = NOT(operand2);

result = operand1 AND operand2; case op of
when
LogicalOp_AND
result = operand1 AND operand2;
when LogicalOp_ORR
result = operand1 OR operand2;

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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## BIF

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the [CPACR\\_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	1	1	Rm				0	0	0	1	1	1	Rn				Rd						
opc2																															

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;
```

## 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.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 =case op of
  when VBitOp_VEOR
    operand1 = V[m];
    operand2 = Zeros();
    operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[d];
operand3 = NOT(operand2 == operand1);
operand3 = NOT(V[m]);

V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);[d] = operand1 EOR ((operand2 EOR operand4) AND
```

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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+00:00

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## BIT

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the [CPACR\\_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	0	1	Rm					0	0	0	1	1	1	Rn					Rd				
opc2																															

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;
```

## 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.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 =case op of
  when VBitOp_VEOR
    operand1 = V[m];
    operand2 = Zeros();
    operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[d];
operand3 = operand2 = operand1;
operand3 = NOT( V[m];[m]);
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);[d] = operand1 EOR ((operand2 EOR operand4) AND
```

## 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)



## BSL

Bitwise Select. This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source 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	0	1	1	Rm				0	0	0	1	1	1	Rn				Rd						
opc2																															

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;
```

## 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.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 =case op of
  when VBitOp_VEOR
    operand1 = V[m];
    operand2 = Zeros();
    operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[m];
operand3 = [d];
    operand2 = operand1;
    operand3 = NOT( V[d];[m]);
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3); [d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## EOR (vector)

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD&FP registers, and places the result in 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	0	0	1	Rm				0	0	0	1	1	1	Rn				Rd						
opc2																															

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize; VBitOp op;

case opc2 of
  when '00' op = VBitOp_VEOR;
  when '01' op = VBitOp_VBSL;
  when '10' op = VBitOp_VBIT;
  when '11' op = VBitOp_VBIF;
```

## 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.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 =case op of
  when VBitOp_VEOR
    operand1 = V[m];
operand2 = Zeros();
operand3 = Ones();
  when VBitOp_VBSL
    operand1 = V[m];
    operand2 = operand1;
    operand3 = V[d];
  when VBitOp_VBIT
    operand1 = V[d];
    operand2 = operand1;
    operand3 = V[m];
  when VBitOp_VBIF
    operand1 = V[d];
    operand2 = operand1;
    operand3 = NOT(V();[m]);
V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

## 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

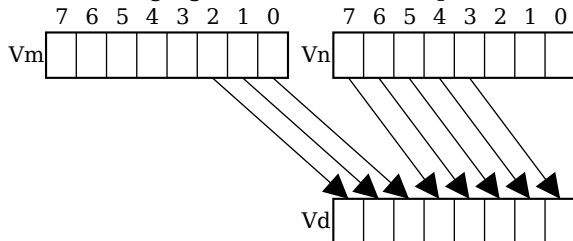
htmldiff from-

(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														Rd

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 = 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;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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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	f	t	y	p	e	1	0	0	0	0	0	1	1	0	0	0	0	Rn				Rd			
																opc																

### Half-precision (ftype == 11) (FEAT\_FP16)

FABS <Hd>, <Hn>

### Single-precision (ftype == 00)

FABS <Sd>, <Sn>

### Double-precision (ftype == 01)

FABS <Dd>, <Dn>

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

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED; FUnaryOp fpop;
case opc of
  when '00' fpop = FUnaryOp_MOV;
  when '01' fpop = FUnaryOp_ABS;
  when '10' fpop = FUnaryOp_NEG;
  when '11' fpop = FUnaryOp_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();

FPCRTType fpcr = FPCR[];
boolean merge = fpcr = FPCR[];
boolean merge = fpop != FUnaryOp_MOV && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

bits(esize) operand = V[n];[n];

case fpop of
  when

FUnaryOp_MOV Elem[result, 0, esize] = [result, 0, esize] = operand;
  when FUnaryOp_ABS Elem[result, 0, esize] = FPAbs(operand);
  when FUnaryOp_NEG Elem[result, 0, esize] = FPNeg(operand);
  when FUnaryOp_SQRT Elem[result, 0, esize] = FPSqrt(operand); (operand, fpcr);
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341  
Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	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	ftype	1	Rm						0	0	1	0	1	0	Rn						Rd					
op																																	

### Half-precision (ftype == 11) (FEAT\_FP16)

FADD <Hd>, <Hn>, <Hm>

### Single-precision (ftype == 00)

FADD <Sd>, <Sn>, <Sm>

### Double-precision (ftype == 01)

FADD <Dd>, <Dn>, <Dm>

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

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 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(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

if sub_op then
Elem[result, 0, esize] = FPSub(operand1, operand2, fpcr);
else
Elem[result, 0, esize] = FPAdd(operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	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	1	0	0	0	0	1	1	0	1	1	0										

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

```
if !HaveFP16Ext() then UNDEFINED;
```

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

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

```
integer esize = 16;
```

```
if sz == '1' then UNDEFINED;
```

```
integer datasize = 32; 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										

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

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

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

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = esize * 2; 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":

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":

<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}; ReduceOp_FADD, operand, esize);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14.5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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.

### Vector

(FEAT\_FCMA)

31	30	29	28	27	26	25	24	23	22	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				

FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

```

if !HaveFCADDExt() then UNDEFINED;
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 << UInt(size);
if !HaveFP16Ext() && 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
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] = FPAAdd(Elem[operand1, e*2, esize], element1, FPCR[]);
  Elem[result, e*2+1, esize] = FPAAdd(Elem[operand1, e*2+1, esize], element3, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41  
Copyright © 2010-2020 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.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands 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	f	t	y	p	e	1	Rm			cond			0	1	Rn			0	nzc			v		
																												op			

### Half-precision (ftype == 11) (FEAT\_FP16)

FCCMP <Hn>, <Hm>, #<nzc>, <cond>

### Single-precision (ftype == 00)

FCCMP <Sn>, <Sm>, #<nzc>, <cond>

### Double-precision (ftype == 01)

FCCMP <Dn>, <Dm>, #<nzc>, <cond>

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

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzc;
```

## 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.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(cond) then
(condition) then
    flags = FPCompare(operand1, operand2, FALSE, FPCR[]);
(operand1, operand2, signal_all_nans, FPCR[]);
PSTATE.<N,Z,C,V> = flags;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the *PSTATE* condition flags to N=0, Z=0, C=1, and V=1.

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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



## FCCMPE

Floating-point Conditional signaling 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.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of 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	f	t	y	p	e	1	Rm				cond				0	1	Rn				1	nzc				v
																												op					

### Half-precision (ftype == 11) (FEAT\_FP16)

FCCMPE <Hn>, <Hm>, #<nzc>, <cond>

### Single-precision (ftype == 00)

FCCMPE <Sn>, <Sm>, #<nzc>, <cond>

### Double-precision (ftype == 01)

FCCMPE <Dn>, <Dm>, #<nzc>, <cond>

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

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

boolean signal_all_nans = (op == '1');
bits(4) condition = cond;
bits(4) flags = nzc;
```

## 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.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(cond) then
(condition) then
    flags = FPCompare(operand1, operand2, TRUE, FPCR[]);
(operand1, operand2, signal_all_nans, FPCR[]);
PSTATE.<N,Z,C,V> = flags;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the *PSTATE* condition flags to N=0, Z=0, C=1, and V=1.

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 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

(FEAT\_FCMA)

31	30	29	28	27	26	25	24	23	22	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													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 = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
if size == '00' || size == '11' then UNDEFINED;
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 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; 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;
FPCRType fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
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]);
[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;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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.

### Vector

(FEAT\_FCMA)

31	30	29	28	27	26	25	24	23	22	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

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

```

if !HaveFCADDExt() then UNDEFINED;
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 << UInt(size);
if !HaveFP16Ext() && 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>
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;
FPCRType fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)





Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;

result = if ConditionHolds(cond) then(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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		ftype		1		0		0		1		0		0		0		0		0		0		0		Rn		Rd	
rmode																opcode																											

**Half-precision to 32-bit (sf == 0 && ftype == 11)**  
(FEAT\_FP16)

FCVTAS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTAS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTAS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTAS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTAS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

FCVTAS <Xd>, <Dn>

```

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if integer fltsize; FPConvOp op;
    FPRounding rounding;
    boolean unsigned;
    integer part;

case ftype 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 HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS() then
      fltsize = 16;
    else
      UNDEFINED;;
  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();

FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, FALSE, fpcr, (fltval, 0, unsigned, fpcr, rounding)); [d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = 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<fsize-1:0>;
    Vpart[d, part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJSFPRounding_TIEAWAYX)(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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	f	t	y	p	e	1	0	0	1	0	1	0	0	0	0	0	0	Rn					Rd	
rmode																opcode															

**Half-precision to 32-bit (sf == 0 && ftype == 11)**  
(FEAT\_FP16)

FCVTAU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTAU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTAU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTAU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTAU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

FCVTAU <Xd>, <Dn>

```

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if integer fltsize; FPConvOp op;
    FPRounding rounding;
    boolean unsigned;
    integer part;

case ftype 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 HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS() then
      fltsize = 16;
    else
      UNDEFINED;;
  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();

FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, TRUE, fpcr, (fltval, 0, unsigned, fpcr, rounding)); [d] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = 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<fsize-1:0>;
    Vpart[d, part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJSFPRounding_TIEAWAYX)(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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	f	t	y	p	e	1	1	0	0	0	0	0	0	0	Rn				Rd					
rmode																opcode															

**Half-precision to 32-bit (sf == 0 && ftype == 11)**  
(FEAT\_FP16)

FCVTMS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTMS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTMS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTMS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTMS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 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 ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  if opcode<2:1>:rmode != '11 01' then UNDEFINED;
  fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = 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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(rmode);
  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();

FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) 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 = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = 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<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, 0, FALSE, fpcr, rounding);(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

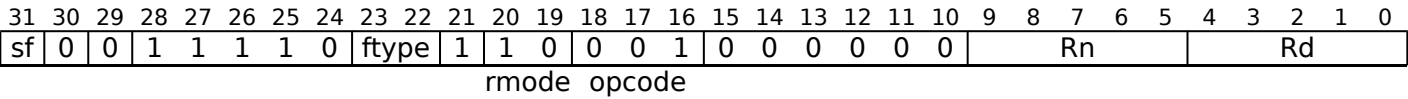
(new)

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 && ftype == 11)**  
(FEAT\_FP16)

FCVTMU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTMU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTMU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTMU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTMU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 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 ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = 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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(rmode);
  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();

FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) 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 = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = 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<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, 0, TRUE, fpcr, rounding);(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

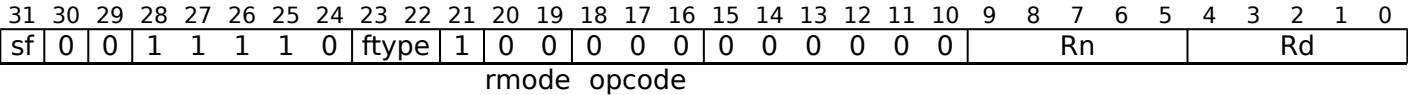
(new)

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 && ftype == 11)**  
(FEAT\_FP16)

FCVTNS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTNS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTNS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTNS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTNS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 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 ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  if opcode<2:1>:rmode != '11 01' then UNDEFINED;
  fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = 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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(rmode);
  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();

FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) 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 = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = 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<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, 0, FALSE, fpcr, rounding);(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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		ftype		1		0		0		0		0		1		0		0		0		0		0		0		Rn		Rd			
rmode																opcode																															

**Half-precision to 32-bit (sf == 0 && ftype == 11)**  
(FEAT\_FP16)

FCVTNU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTNU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTNU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTNU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTNU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 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 ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = 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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(rmode);
  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();

FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) 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 = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = 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<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, 0, TRUE, fpcr, rounding);(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

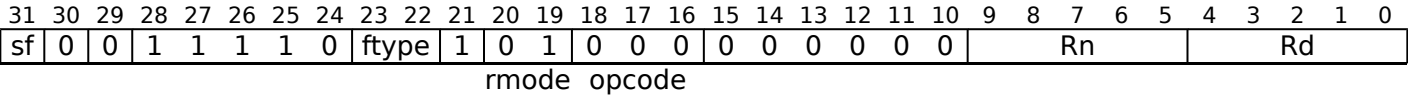
(new)

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 && ftype == 11)**  
(FEAT\_FP16)

FCVTPS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTPS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTPS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTPS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTPS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 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 ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = 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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(rmode);
  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();

FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) 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 = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = 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<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, 0, FALSE, fpcr, rounding);(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53.41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

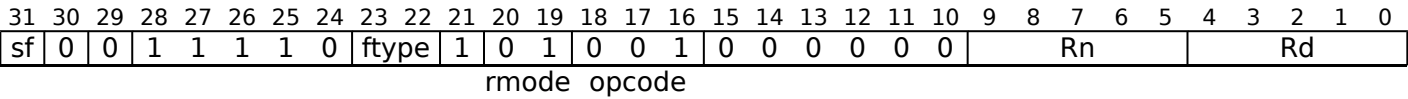
(new)

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 && ftype == 11)**  
(FEAT\_FP16)

FCVTPU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTPU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTPU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTPU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTPU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 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 ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = 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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(rmode);
  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();

FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPCnvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when FPCnvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, 0, unsigned, fpcr, rounding);
  X[d] = intval;
  when FPCnvOp_CVT_ItoF
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, unsigned, fpcr, rounding);
    V[d] = fltval;
  when FPCnvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPCnvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPCnvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, 0, TRUE, fpcr, rounding);(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

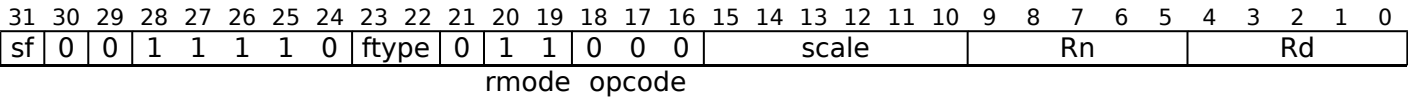
(new)

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 && ftype == 11)**  
(FEAT\_FP16)

FCVTZS <Wd>, <Hn>, #<fbits>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTZS <Xd>, <Hn>, #<fbits>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTZS <Wd>, <Sn>, #<fbits>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTZS <Xd>, <Sn>, #<fbits>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTZS <Wd>, <Dn>, #<fbits>

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

FCVTZS <Xd>, <Dn>, #<fbits>

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if integer fltsize; FPConvOp op;
    FPRounding rounding;
    boolean unsigned;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
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(scale);
  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> 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".  
  
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".

Operation

```
CheckFPAdvSIMDEnabled64();
FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, fracbits, FALSE, fpcr, {fltval, fracbits, unsigned, fpcr, rounding}); FPRounding
  when
    FPConvOp_CVT_ItoF
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, unsigned, fpcr, rounding);
    V[d] = intval;[d] = fltval;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341  
Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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		0		0		0		0		0		0		0		0		0	
rmode																opcode																	

**Half-precision to 32-bit (sf == 0 && ftype == 11)**  
(FEAT\_FP16)

FCVTZS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTZS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTZS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTZS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTZS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 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 ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  if opcode<2:1>:rmode != '11 01' then UNDEFINED;
  fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = 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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(rmode);
  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();

FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) 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 = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = 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<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, 0, FALSE, fpcr, rounding);(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 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		ftype		0		1		1		0		0		1		scale				Rn				Rd			
rmode																opcode																									

**Half-precision to 32-bit (sf == 0 && ftype == 11)**  
(FEAT\_FP16)

FCVTZU <Wd>, <Hn>, #<fbits>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTZU <Xd>, <Hn>, #<fbits>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTZU <Wd>, <Sn>, #<fbits>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTZU <Xd>, <Sn>, #<fbits>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTZU <Wd>, <Dn>, #<fbits>

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

FCVTZU <Xd>, <Dn>, #<fbits>

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if integer fltsize; FPConvOp op;
    FPRounding rounding;
    boolean unsigned;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
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(scale);
  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> 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".  
  
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".

Operation

```
CheckFPAdvSIMDEnabled64();
FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPCConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when FPCConvOp_CVT_FtoI
    fltval = V[n];
  intval = FPToFixed(fltval, fracbits, TRUE, fpcr, (fltval, fracbits, unsigned, fpcr, rounding); FPRounding
  when
    FPCConvOp_CVT_ItoF
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, unsigned, fpcr, rounding);
    V[d] = intval;[d] = fltval;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341  
Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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		0		1		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0	

**Half-precision to 32-bit (sf == 0 && ftype == 11)**  
(FEAT\_FP16)

FCVTZU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

FCVTZU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTZU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTZU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTZU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 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 ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = 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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(rmode);
  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();

FPCRTType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) 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 = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = 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<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, 0, TRUE, fpcr, rounding);(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

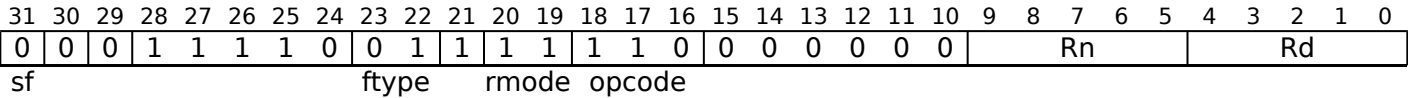
htmldiff from-

(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 represented 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 (FEAT\_JSCVT)





FJCVTZS <Wd>, <Dn>

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

if !integer intsize = if sf == '1' then 64 else 32;
integer fltsize; FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case ftype 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 HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS() then UNDEFINED;
  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();

FPCRTType fpcr = FPCR[];
bits(64) fltval;
bits(32) intval;

bit Z;
fltval =boolean merge = IsMerging(fpcr);
integer fsize = if op == FPCnvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when FPCnvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, fpcr, rounding);
    X[d] = intval;
  when FPCnvOp_CVT_ItoF
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, unsigned, fpcr, rounding);
    V[d] = fltval;
  when FPCnvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPCnvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPCnvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	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		ftype		0		Rm						0		Ra						Rn						Rd					
										o1						o0																													

### Half-precision (ftype == 11) (FEAT\_FP16)

FMADD <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (ftype == 00)

FMADD <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (ftype == 01)

FMADD <Dd>, <Dn>, <Dm>, <Da>

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

```
integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
```

UNDEFINED; 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(esize) operanda = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();();

if opa_neg then operanda =
FPNeg(operanda);
if opl_neg then operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d] = result;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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	f	t	y	p	e	1	Rm			0	1	0	0	1	0	Rn			Rd					
op																															

### Half-precision (ftype == 11) (FEAT\_FP16)

FMAX <Hd>, <Hn>, <Hm>

### Single-precision (ftype == 00)

FMAX <Sd>, <Sn>, <Sm>

### Double-precision (ftype == 01)

FMAX <Dd>, <Dn>, <Dm>

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

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED; FPMaMinOp operation;
case op of
    when '00' operation = FPMaMinOp_MAX;
    when '01' operation = FPMaMinOp_MIN;
    when '10' operation = FPMaMinOp_MAXNUM;
    when '11' operation = FPMaMinOp_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(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

case operation of
  when

FPMaMinOp_MAX Elem[result, 0, esize] = FPMa(operand1, operand2, fpcr);
  when FPMaMinOp_MIN Elem[result, 0, esize] = FPMi(operand1, operand2, fpcr);
  when FPMaMinOp_MAXNUM Elem[result, 0, esize] = FPMaNum(operand1, operand2, fpcr);
  when FPMaMinOp_MINNUM Elem[result, 0, esize] = FPMiNum(operand1, operand2, fpcr);
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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	ftype	1	Rm						0	1	1	0	1	0	Rn						Rd					
op																																	

### Half-precision (ftype == 11) (FEAT\_FP16)

FMAXNM <Hd>, <Hn>, <Hm>

### Single-precision (ftype == 00)

FMAXNM <Sd>, <Sn>, <Sm>

### Double-precision (ftype == 01)

FMAXNM <Dd>, <Dn>, <Dm>

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

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED; FPMaMinOp operation;
case op of
    when '00' operation = FPMaMinOp_MAX;
    when '01' operation = FPMaMinOp_MIN;
    when '10' operation = FPMaMinOp_MAXNUM;
    when '11' operation = FPMaMinOp_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(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

case operation of
  when

FPMaMinOp_MAX Elem[result, 0, esize] = FPMa(operand1, operand2, fpcr);
  when FPMaMinOp_MIN Elem[result, 0, esize] = FPMi(operand1, operand2, fpcr);
  when FPMaMinOp_MAXNUM Elem[result, 0, esize] = FPMaNum(operand1, operand2, fpcr);
  when FPMaMinOp_MINNUM Elem[result, 0, esize] = FPMiNum(operand1, operand2, fpcr);
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	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	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
o1																															

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

```
if !HaveFP16Ext() then UNDEFINED;
```

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

```
integer esize = 16;
```

```
if sz == '1' then UNDEFINED;
```

```
integer datasize = 32; 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																						

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

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

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = esize * 2; 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];[n];
boolean altfp = FALSE;
V[d] = Reduce({op, operand, esize, altfp};ReduceOp_FMAXNUM, operand, esize, FALSE);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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 (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	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																															

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

```
if !HaveFP16Ext() then UNDEFINED;
```

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

```
integer esize = 16;
```

```
integer datasize = if Q == '1' then 128 else 64; 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																															

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

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

```
if sz:Q != '01' then UNDEFINED;    // .4S only
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = if Q == '1' then 128 else 64; 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];[n];
boolean altfp = FALSE;
V[d] = Reduce({op, operand, esize, altfp};ReduceOp_FMAXNUM, operand, esize, FALSE);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	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	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
o1																															

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

```
if !HaveFP16Ext() then UNDEFINED;
```

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

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

```
integer esize = 16;
```

```
if sz == '1' then UNDEFINED;
```

```
integer datasize = 32; 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																															

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

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

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

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = esize * 2; 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”:	
sz		<T>	
0		2S	
1		2D	

Operation

```

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

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	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																															

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

```
if !HaveFP16Ext() then UNDEFINED;
```

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

```
integer esize = 16;
```

```
integer datasize = if Q == '1' then 128 else 64; 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																															

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

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

```
if sz:Q != '01' then UNDEFINED;
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = if Q == '1' then 128 else 64; 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}; ReduceOp_FMAX, operand, esize);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: ~~2020-12-16T16:53:41~~ 2020-12-16T14:53:41

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	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	ftype	1	Rm				0	1	0	1	1	0	Rn				Rd							
op																															

### Half-precision (ftype == 11) (FEAT\_FP16)

FMIN <Hd>, <Hn>, <Hm>

### Single-precision (ftype == 00)

FMIN <Sd>, <Sn>, <Sm>

### Double-precision (ftype == 01)

FMIN <Dd>, <Dn>, <Dm>

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

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 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(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

case operation of
  when

FPMaMinOp_MAX Elem[result, 0, esize] = FPMa(operand1, operand2, fpcr);
  when FPMaMinOp_MIN Elem[result, 0, esize] = FPMi(operand1, operand2, fpcr);
  when FPMaMinOp_MAXNUM Elem[result, 0, esize] = FPMaNum(operand1, operand2, fpcr);
  when FPMaMinOp_MINNUM Elem[result, 0, esize] = FPMiNum(operand1, operand2, fpcr);
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	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	ftype			1	Rm				0	1	1	1	1	0	Rn				Rd					
op																															

### Half-precision (ftype == 11) (FEAT\_FP16)

FMINNM <Hd>, <Hn>, <Hm>

### Single-precision (ftype == 00)

FMINNM <Sd>, <Sn>, <Sm>

### Double-precision (ftype == 01)

FMINNM <Dd>, <Dn>, <Dm>

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

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 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(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();();

case operation of
  when

FPMaMinOp_MAX    Elem[result, 0, esize] = FPMa(operand1, operand2, fpcr);
  when FPMaMinOp_MIN    Elem[result, 0, esize] = FPMi(operand1, operand2, fpcr);
  when FPMaMinOp_MAXNUM Elem[result, 0, esize] = FPMaN(operand1, operand2, fpcr);
  when FPMaMinOp_MINNUM Elem[result, 0, esize] = FPMiN(operand1, operand2, fpcr);

V[d] = result;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	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	1	0	0	0	0	1	1	0	0	1	0	Rn						Rd					
o1																																	

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

```
if !HaveFP16Ext() then UNDEFINED;
```

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

```
integer esize = 16;
```

```
if sz == '1' then UNDEFINED;
```

```
integer datasize = 32; 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																															

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

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

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = esize * 2; 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];[n];
boolean altfp = FALSE;
V[d] = Reduce({op, operand, esize, altfp};ReduceOp_FMINNUM, operand, esize, FALSE);

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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 (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	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																							

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

```
if !HaveFP16Ext() then UNDEFINED;
```

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

```
integer esize = 16;
```

```
integer datasize = if Q == '1' then 128 else 64; integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize; ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMA
```

### 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																							

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

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

```
if sz:Q != '01' then UNDEFINED;    // .4S only
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = if Q == '1' then 128 else 64; integer datasize = if Q == '1' then 128 else 64;
```

```
integer elements = datasize DIV esize; ReduceOp op = if o1 == '1' then ReduceOp_FMINNUM else ReduceOp_FMA
```

### 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];[n];
boolean altfp = FALSE;
V[d] = Reduce({op, operand, esize, altfp};ReduceOp_FMINNUM, operand, esize, FALSE);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## 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 (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	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	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
ol																															

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

```
if !HaveFP16Ext() then UNDEFINED;
```

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

```
integer esize = 16;
```

```
if sz == '1' then UNDEFINED;
```

```
integer datasize = 32; integer datasize = esize * 2;
```

```
integer elements = 2; ReduceOp op = if ol == '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																						

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

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

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = esize * 2; integer datasize = esize * 2;
```

```
integer elements = 2; ReduceOp op = if ol == '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”:

sz	<T>
0	2S
1	2D

Operation

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

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	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																							

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

```
if !HaveFP16Ext() then UNDEFINED;
```

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

```
integer esize = 16;
```

```
integer datasize = if Q == '1' then 128 else 64; 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																							

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

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

```
if sz:Q != '01' then UNDEFINED;
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = if Q == '1' then 128 else 64; 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};ReduceOp_FMIN, operand, esize);

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: ~~2020-12-16T16:53:41~~2020-12-16T14:53:41

Copyright © 2010-2020 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 (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	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	L	M				Rm		0	0	0	0	H	0								Rd	
SZ												S																			

FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
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 UNDEFINED;
integer index = UInt(H:L:M);
(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer datasize = if Q=='1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;

```

### FMLAL2 (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	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	L	M				Rm		1	0	0	0	H	0								Rd	
SZ												S																			

FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
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 UNDEFINED;
integer index = UInt(H:L:M);
(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
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];
[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## 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 (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	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	Rm				1	1	1	0	1	1	Rn				Rd						
S																sz															

FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(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 (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	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	1	Rm				1	1	0	0	1	1	Rn				Rd						
S																sz															

FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(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];
[n, part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
[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([result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, e
V[d] = result;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14Z

Copyright © 2010-2020 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 (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	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	L	M			Rm		0	1	0	0	H	0									Rd	
SZ												S																			

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
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 UNDEFINED;
integer index = UInt(H:L:M);
(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer datasize = if Q=='1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;

```

### FMLSL2 (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	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	L	M			Rm		1	1	0	0	H	0									Rd	
SZ												S																			

```
FMLSLS2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
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 UNDEFINED;
integer index = UInt(H:L:M);
(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
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];
[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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 (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	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	Rm				1	1	1	0	1	1	Rn				Rd						
S																sz															

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(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 (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	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	0	1	Rm				1	1	0	0	1	1	Rn				Rd						
								S		sz																					

FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(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];
[n, part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
[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([result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, e
V[d] = result;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	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				

FMOV <Vd>.<T>, #<imm>

```
if !HaveFP16Ext() then UNDEFINED;
```

```
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>:(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;

if cmode:op == '11111' then
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;

imm64 =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 = 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;
```

(old)

htmldiff from-

(new)

## 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	ftype				1	0	0	0	0	0	1	0	0	0	0	Rn					Rd				
opc																																

### Half-precision (ftype == 11) (FEAT\_FP16)

FMOV <Hd>, <Hn>

### Single-precision (ftype == 00)

FMOV <Sd>, <Sn>

### Double-precision (ftype == 01)

FMOV <Dd>, <Dn>

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

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED; FUnaryOp fpop;
case opc of
  when '00' fpop = FUnaryOp_MOV;
  when '01' fpop = FUnaryOp_ABS;
  when '10' fpop = FUnaryOp_NEG;
  when '11' fpop = FUnaryOp_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(esize) operand =(); FPCRTYPE fpcr = FPCR[];
boolean merge = fpop != FPUUnaryOp_MOV && IsMerging(fpcr);
bits(128) result = if merge then V[n];[d] else

Zeros();

bits(esize) operand = V[n];

case fpop of
  when FPUUnaryOp_MOV Elem[result, 0, esize] = operand;
  when ZerosFPUUnaryOp_ABS(), 0, esize] = operand;
Elem[result, 0, esize] = FPAbs(operand);
  when FPUUnaryOp_NEG Elem[result, 0, esize] = FPNeg(operand);
  when FPUUnaryOp_SQRT Elem[result, 0, esize] = FPSqrt(operand, fpcr);

V[d] = Zeros();[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14.5341

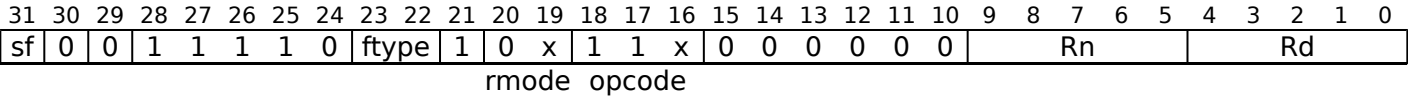
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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.



**Half-precision to 32-bit (sf == 0 && ftype == 11 && rmode == 00 && opcode == 110)**  
(FEAT\_FP16)

FMOV <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11 && rmode == 00 && opcode == 110)**  
(FEAT\_FP16)

FMOV <Xd>, <Hn>

**32-bit to half-precision (sf == 0 && ftype == 11 && rmode == 00 && opcode == 111)**  
(FEAT\_FP16)

FMOV <Hd>, <Wn>

**32-bit to single-precision (sf == 0 && ftype == 00 && rmode == 00 && opcode == 111)**

FMOV <Sd>, <Wn>

**Single-precision to 32-bit (sf == 0 && ftype == 00 && rmode == 00 && opcode == 110)**

FMOV <Wd>, <Sn>

**64-bit to half-precision (sf == 1 && ftype == 11 && rmode == 00 && opcode == 111)**  
(FEAT\_FP16)

FMOV <Hd>, <Xn>

**64-bit to double-precision (sf == 1 && ftype == 01 && rmode == 00 && opcode == 111)**

FMOV <Dd>, <Xn>

**64-bit to top half of 128-bit (sf == 1 && ftype == 10 && rmode == 01 && opcode == 111)**

FMOV <Vd>.D[1], <Xn>

**Double-precision to 64-bit (sf == 1 && ftype == 01 && rmode == 00 && opcode == 110)**

FMOV <Xd>, <Dn>

**Top half of 128-bit to 64-bit (sf == 1 && ftype == 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 ftype 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 HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS;
  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();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) 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 = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, unsigned, fpcr, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n, part];
[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fsize-1:0>;
    Vpart[d, part] = fltval;
[d,part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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	ftype	0	Rm					1	Ra					Rn					Rd										
										o1																	o0									

### Half-precision (ftype == 11) (FEAT\_FP16)

FMSUB <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (ftype == 00)

FMSUB <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (ftype == 01)

FMSUB <Dd>, <Dn>, <Dm>, <Da>

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

```
integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
```

UNDEFINED; 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(esize) operandA = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operand1 = if opa_neg then operandA = FPNeg(operand1);(operandA);
if opl_neg then operand1 =
FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operandA, operand1, operand2, fpcr);

V[d] = result;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53 2020-12-16T14:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

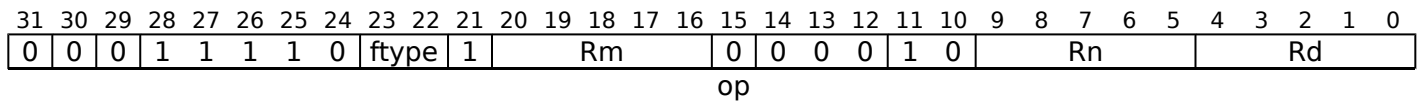
(new)

## 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.



### Half-precision (ftype == 11) (FEAT\_FP16)

FMUL <Hd>, <Hn>, <Hm>

### Single-precision (ftype == 00)

FMUL <Sd>, <Sn>, <Sm>

### Double-precision (ftype == 01)

FMUL <Dd>, <Dn>, <Dm>

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

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 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(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

bits(esize) product = FPMul(operand1, operand2, fpcr);
if negated then product =
FPNeg(product);
Elem[result, 0, esize] = product;

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(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	f	t	y	p	e	1	0	0	0	0	1	0	1	0	0	0	0	Rn				Rd			
opc																																

### Half-precision (ftype == 11) (FEAT\_FP16)

FNEG <Hd>, <Hn>

### Single-precision (ftype == 00)

FNEG <Sd>, <Sn>

### Double-precision (ftype == 01)

FNEG <Dd>, <Dn>

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

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED; FUnaryOp fpop;
case opc of
  when '00' fpop = FUnaryOp_MOV;
  when '01' fpop = FUnaryOp_ABS;
  when '10' fpop = FUnaryOp_NEG;
  when '11' fpop = FUnaryOp_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();

FPCRTType fpcr = FPCR[];
boolean merge = fpcr = FPCR[];
boolean merge = fpop != FPUUnaryOp_MOV && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

bits(esize) operand = V[n];[n];

case fpop of
  when

FPUUnaryOp_MOV Elem[result, 0, esize] = [result, 0, esize] = operand;
  when FPUUnaryOp_ABS Elem[result, 0, esize] = FPAbs(operand);
  when FPUUnaryOp_NEG Elem[result, 0, esize] = FPNeg(operand);
  when FPUUnaryOp_SQRT Elem[result, 0, esize] = FPSqrt(operand); (operand, fpcr);
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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	ftype	1	Rm					0	Ra					Rn					Rd					
										o1										o0											

### Half-precision (ftype == 11) (FEAT\_FP16)

FNMADD <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (ftype == 00)

FNMADD <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (ftype == 01)

FNMADD <Dd>, <Dn>, <Dm>, <Da>

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

```
integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
```

UNDEFINED; 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(esize) operanda = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operanda =if opa_neg then operanda = FPNeg(operanda);
operand1 =if opl_neg then operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d] = result;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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	ftype	1	Rm					1	Ra					Rn					Rd										
										o1																	o0									

### Half-precision (ftype == 11) (FEAT\_FP16)

FNMSUB <Hd>, <Hn>, <Hm>, <Ha>

### Single-precision (ftype == 00)

FNMSUB <Sd>, <Sn>, <Sm>, <Sa>

### Double-precision (ftype == 01)

FNMSUB <Dd>, <Dn>, <Dm>, <Da>

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

```
integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
```

UNDEFINED; 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(esize) operanda = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operanda = if opa_neg then operanda = FPNeg(operanda);
operand1 = if opl_neg then operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d] = result;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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	f <sub>type</sub>	1	Rm				1	0	0	0	1	0	Rn				Rd							
op																															

**Half-precision (ftype == 11)**  
(FEAT\_FP16)

FNMUL <Hd>, <Hn>, <Hm>

### Single-precision (ftype == 00)

FNMUL <Sd>, <Sn>, <Sm>

### Double-precision (ftype == 01)

FNMUL <Dd>, <Dn>, <Dm>

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

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 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(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

bits(esize) product = FPMul(operand1, operand2, fpcr);
product =if negated then product = FPNeg(product);
Elem[result, 0, esize] = product;

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	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 (FEAT\_FRINTTS)

31	30	29	28	27	26	25	24	23	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					
ftype										op																							

#### Single-precision (ftype == 00)

FRINT32X <Sd>, <Sn>

#### Double-precision (ftype == 01)

FRINT32X <Dd>, <Dn>

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED; when '1x' UNDEFINED;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = 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();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, rounding, 32);(operand, fpcr, rounding, intsize);

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	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 (FEAT\_FRINTTS)

31	30	29	28	27	26	25	24	23	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			
ftype										op																					

#### Single-precision (ftype == 00)

FRINT32Z <Sd>, <Sn>

#### Double-precision (ftype == 01)

FRINT32Z <Dd>, <Dn>

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
```

```
integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED; when '1x' UNDEFINED;
```

```
integer intsize = if op<1> == '0' then 32 else 64; FPRounding rounding = if op<0> == '0' then FPRounding_2
```

### 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();

FPCRTyp fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, {operand, fpcr, rounding, intsize}; FPRounding_ZERO, 3
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	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 (FEAT\_FRINTTS)

31	30	29	28	27	26	25	24	23	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					
ftype										op																					

#### Single-precision (ftype == 00)

FRINT64X <Sd>, <Sn>

#### Double-precision (ftype == 01)

FRINT64X <Dd>, <Dn>

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED; when '1x' UNDEFINED;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = 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();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, rounding, 64);(operand, fpcr, rounding, intsize);

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	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 (FEAT\_FRINTTS)

31	30	29	28	27	26	25	24	23	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			
ftype										op																					

#### Single-precision (ftype == 00)

FRINT64Z <Sd>, <Sn>

#### Double-precision (ftype == 01)

FRINT64Z <Dd>, <Dn>

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
```

```
integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED; when '1x' UNDEFINED;
```

```
integer intsize = if op<1> == '0' then 32 else 64; FPRounding rounding = if op<0> == '0' then FPRounding_2
```

### 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();

FPCRTyp fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, operand, fpcr, rounding, intsize); FPRounding_ZERO, 0
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53 2020-12-16T14:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## 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	f	t	y	p	e	1	0	0	1	1	0	0	0	0	0	Rn				Rd				
rmode																															

### Half-precision (ftype == 11) (FEAT\_FP16)

FRINTA <Hd>, <Hn>

### Single-precision (ftype == 00)

FRINTA <Sd>, <Sn>

### Double-precision (ftype == 01)

FRINTA <Dd>, <Dn>

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

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
      UNDEFINED;

boolean exact = FALSE; FPRounding rounding;
case rmode of
  when '0xx' rounding = FPDecodeRounding(rmode<1:0>);
  when '100' rounding = FPRounding_TIEAWAY;
  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();

FPCRTyp fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, operand, fpcr, rounding, exact); FPRounding_TIEAWAY, P
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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	f	t	y	p	e	1	0	0	1	1	1	1	0	0	0	0	Rn				Rd			
rmode																															

### Half-precision (ftype == 11) (FEAT\_FP16)

FRINTI <Hd>, <Hn>

### Single-precision (ftype == 00)

FRINTI <Sd>, <Sn>

### Double-precision (ftype == 01)

FRINTI <Dd>, <Dn>

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

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
    when '0xx' rounding = FPCDecodeRounding(rmode<1:0>);
    when '100' rounding = FPRounding_TIEAWAY;
    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();  
  
FPCRTyp fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d] else Zeros();  
bits(esize) operand = V[n];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);  
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41  
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## 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	f	t	y	p	e	1	0	0	1	0	1	0	0	0	0	Rn				Rd				
																rmode															

### Half-precision (ftype == 11) (FEAT\_FP16)

FRINTM <Hd>, <Hn>

### Single-precision (ftype == 00)

FRINTM <Sd>, <Sn>

### Double-precision (ftype == 01)

FRINTM <Dd>, <Dn>

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

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPCDecodeRounding('10');
  when '100' rounding = FPRounding_TIEAWAY;
  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();  
  
FPCRTyp fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d] else Zeros();  
bits(esize) operand = V[n];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);  
  
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41  
Copyright © 2010-2020 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	f	t	y	p	e	1	0	0	1	0	0	0	1	0	0	0	0	Rn			Rd			
rmode																															

### Half-precision (ftype == 11) (FEAT\_FP16)

FRINTN <Hd>, <Hn>

### Single-precision (ftype == 00)

FRINTN <Sd>, <Sn>

### Double-precision (ftype == 01)

FRINTN <Dd>, <Dn>

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

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPCDecodeRounding('00');
  when '100' rounding = FPRounding_TIEAWAY;
  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();  
  
FPCRTyp fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d] else Zeros();  
bits(esize) operand = V[n];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);  
  
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41  
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## 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	f	t	y	p	e	1	0	0	1	0	0	0	0	0	Rn					Rd				
																	rmode														

### Half-precision (ftype == 11) (FEAT\_FP16)

FRINTP <Hd>, <Hn>

### Single-precision (ftype == 00)

FRINTP <Sd>, <Sn>

### Double-precision (ftype == 01)

FRINTP <Dd>, <Dn>

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

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPCDecodeRounding('01');
  when '100' rounding = FPRounding_TIEAWAY;
  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();

FPCRTyp fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41  
Copyright © 2010-2020 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 the result value is not numerically equal to the 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.

31	30	29	28	27	26	25	24	23	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	f	t	y	p	e	1	0	0	1	1	1	0	1	0	0	0	0	Rn					Rd				
rmode																																		

### Half-precision (ftype == 11) (FEAT\_FP16)

FRINTX <Hd>, <Hn>

### Single-precision (ftype == 00)

FRINTX <Sd>, <Sn>

### Double-precision (ftype == 01)

FRINTX <Dd>, <Dn>

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

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
      UNDEFINED;
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPCodeRounding(rmode<1:0>);
  when '100' rounding = FPRounding_TIEAWAY;
  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();  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d] else Zeros();  
bits(esize) operand = V[n];  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, TRUE);  
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41  
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## 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	f	t	y	p	e	1	0	0	1	0	1	1	1	0	0	0	0	Rn					Rd				
rmode																																		

### Half-precision (ftype == 11) (FEAT\_FP16)

FRINTZ <Hd>, <Hn>

### Single-precision (ftype == 00)

FRINTZ <Sd>, <Sn>

### Double-precision (ftype == 01)

FRINTZ <Dd>, <Dn>

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

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
      UNDEFINED;
boolean exact = FALSE;

FPRounding rounding;
rounding = case rmode of
  when '0xx' rounding = FPCDecodeRounding('11'); (rmode<1:0>);
  when '100' rounding = FPRounding_TIEAWAY;
  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();  
  
FPCRTyp fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d] else Zeros();  
bits(esize) operand = V[n];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);  
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41  
Copyright © 2010-2020 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	ftype		1	0	0	0	0	1	1	1	0	0	0	0	Rn					Rd				
opc																															

### Half-precision (ftype == 11) (FEAT\_FP16)

FSQRT <Hd>, <Hn>

### Single-precision (ftype == 00)

FSQRT <Sd>, <Sn>

### Double-precision (ftype == 01)

FSQRT <Dd>, <Dn>

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

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED; FPUUnaryOp fpop;
case opc of
    when '00' fpop = FPUUnaryOp_MOV;
    when '01' fpop = FPUUnaryOp_ABS;
    when '10' fpop = FPUUnaryOp_NEG;
    when '11' fpop = FPUUnaryOp_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();

FPCRTyp fpcr = FPCR[];
boolean merge = fpcr == FPCR[];
boolean merge = fpcr != FPUUnaryOp_MOV && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

bits(esize) operand = V[n];

case fpop of
  when FPUUnaryOp_MOV Elem[result, 0, esize] = operand;
  when FPUUnaryOp_ABS Elem[result, 0, esize] = FPAbs(operand);
  when FPUUnaryOp_NEG Elem[result, 0, esize] = FPNeg(operand);
  when FPUUnaryOp_SQRT[n];

Elem[result, 0, esize] = FPSqrt(operand, fpcr);

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41  
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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



Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

if sub_op then
Elem[result, 0, esize] = FPSub(operand1, operand2, fpcr);
else
Elem[result, 0, esize] = FPAdd(operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 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\\_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 \(from general\)](#).

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				

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 = UInt(imm5<4:size+1>);

integer esize = 8 << size; 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
xxx1	B
xx10	H
xx100	S
x1000	D

<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>

<R> Is the width specifier for the general-purpose source register, encoded in "imm5":

imm5	<R>
x0000	RESERVED
xxx1	W
xx10	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(128) result;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

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;

// .LD format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: ~~2020-12-16T16:20:14~~2020-12-16T14:53:41

Copyright © 2010-2020 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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

**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<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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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

31	30	29	28	27	26	25	24	23	22	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	0	0	size	Rn					Rt					
L R										opcode S																					

LD1R { <Vt>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### 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	0	0	size	Rn					Rt					
L R										opcode S																					

### 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;
boolean tag_checked = wback || n != 31;
```

## 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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																					

LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

## 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;

// .LD format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

**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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																					

LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

### 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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

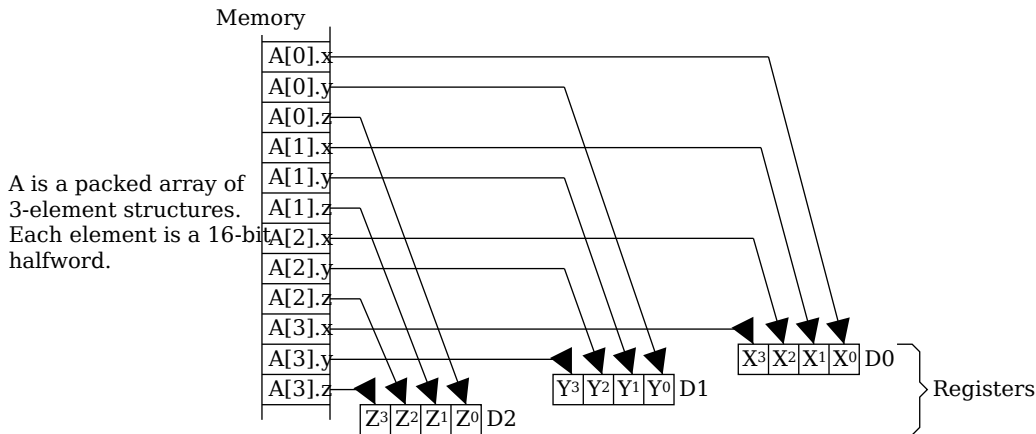
htmldiff from-

(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																					

LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

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;

// .LD format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: ~~2020-12-16T16:2020-12-16T14:5341~~

Copyright © 2010-2020 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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

**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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																					

LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

## 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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																					

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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

## 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;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: ~~2020-12-16T16:20:14~~2020-12-16T14:53:41

Copyright © 2010-2020 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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

## 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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the 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	1	0	1	1	0	0	0	0	0	1	1	1	0	size	Rn					Rt					
L R										opcode S																					

LD4R { <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;
boolean tag_checked = wback || n != 31;
```

### 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	1	0	size	Rn					Rt					
L										R	opcode										S										

### Immediate offset (Rm == 11111)

LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

### Register offset (Rm != 11111)

LD4R { <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;
boolean tag_checked = wback || n != 31;
```

### 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> 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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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	1	imm7							Rt2					Rn				Rt					
										L																					

### 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>}]
```

```
// Empty.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);
if opc == '11' then UNDEFINED;
integer scale = 2 + (Rt2); AccType acctype = AccType_VECSTREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if memop == MemOp_LOAD(imm7, 64), scale);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then && 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 Constraint_NOP      EndOfInstruction();

```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data1 = case memop of
    when MemOp_STORE
        data1 = V[t];
        data2 = V[t2];
        Mem[address, dbytes, [address + 0, dbytes, acctype] = data1; AccType_VECSTREAMMem];
data2 = [address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype];
        data2 = AccType_VECSTREAMMem];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN; [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[t2] = data2; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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								

#### 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						
										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 = 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);
if opc == '11' then UNDEFINED;
integer scale = 2 + (Rt2); AccType acctype = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if memop == MemOp_LOAD(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if t == t2 then && 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 Constraint_NOP     EndOfInstruction();
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

data1 = case memop of
when MemOp_STORE
    data1 = V[t];
    data2 = V[t2];
    Mem[address, dbytes, [address + 0, dbytes, acctype] = data1; AccType_VECMem];
data2 = [address + dbytes, dbytes, acctype] = data2;

when MemOp_LOAD
    data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype];
    data2 = AccType_VECMem];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 = 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)

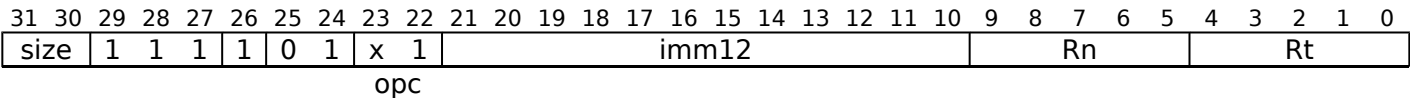
```
LDR <Dt>, [<Xn|SP>, #<sim>]!
```

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 = 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 = 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.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	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.  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 <pimm>/2.  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.  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 <pimm>/16.

## 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;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```



Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype] = data;
when AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 = SignExtend(imm19:'00', 64);(imm19:'00', 64);
boolean tag_checked = FALSE;
```

## 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;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);(tag-checked);

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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 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;
boolean tag_checked = memop != MemOp_PREFETCH;
```

## Operation

```

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype] = data;

    when AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype]; AccType_VECV]; [t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        [] = address;
    else
        XVSP[t] = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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 = 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;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31); && (wback || n != 31);
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype]] = data;

    when AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype]; AccType_VECV]; [t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        [] = address;
    else
        XVSP[t] = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	22	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	0	0	H	0	Rn				Rd						

o2

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 = 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>;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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.

31	30	29	28	27	26	25	24	23	22	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	1	0	1	Rn						Rd			
U																															

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 << 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)*(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

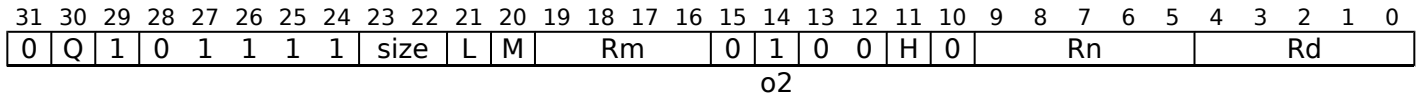
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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.



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 = 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(idxdsiize) 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>;
    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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	22	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	1	0	1	Rn						Rd			
U																															

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 << 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)*(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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								Rd		

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 = 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

<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 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>; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	22	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	1	1	1	Rn						Rd				
U																																

U

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 << 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)*(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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 '0xx01' operation = when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx11' operation = when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x01' operation = when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x1' operation = when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '1110x' operation = when '11110' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, 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.

31	30	29	28	27	26	25	24	23	22	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	Rm					0	0	0	1	1	1	Rn					Rd				
size																															

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

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

boolean invert = (size<0> == '1'); LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

## 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.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

operand2 = NOT(operand2);
if invert then operand2 = NOT(operand2);

result = operand1 OR operand2; case op of
  when
    LogicalOp_AND
      result = operand1 AND operand2;
  when LogicalOp_ORR
      result = operand1 OR operand2;

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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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 '0xx10' operation = when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '10x00' operation = when '0xx11' operation = ImmediateOp_BIC;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x10' operation = 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 '110x0' operation = when '1110x' operation = ImmediateOp_MOVI;
  when '1110x' operation = when '11110' operation = ImmediateOp_MOVI;
  when '11110' operation = when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    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>”:

<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>":

<b>cmode&lt;2:1&gt;</b>	<b>&lt;amount&gt;</b>
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14.5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

## htmldiff from-

(new)

## ORR (vector, register)

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, 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 \(vector\)](#).

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	1	0	1	Rm					0	0	0	1	1	1	Rn					Rd				
size																															

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64; integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean invert = (size<0> == '1'); LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

### 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.

### Alias Conditions

Alias	Is preferred when
<a href="#">MOV (vector)</a>	Rm == Rn

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
result = operand1 OR operand2; if invert then operand2 = NOT(operand2);
case op of
  when
    LogicalOp_AND
      result = operand1 AND operand2;
  when LogicalOp_ORR
      result = operand1 OR operand2;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	22	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	1	1	1	Rn						Rd			
U																															

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 << 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)*(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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 [FEAT\\_SHA3](#) is implemented.

### Advanced SIMD (FEAT\_SHA3)

31	30	29	28	27	26	25	24	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						

RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D

```
if !HaveSHA3Ext() then UNDEFINED;
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):(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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	1	1	0	Rn				Rd					
U										o0																					

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) + (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;
```

## 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 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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	1	0	Rn				Rd				
U										o0																					

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) + (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;
```

## 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																					

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) + (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;
```

## 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	22	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	1	1	Rn						Rd			
U											ac																				

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 << 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>; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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																							

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 << 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>; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	22	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	1	Rn						Rd			
U											ac																				

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 << 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>; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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																							

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 << 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>; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

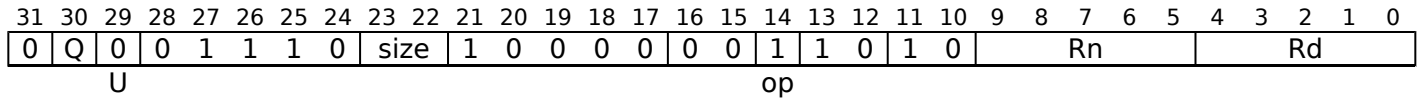
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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.



SADALP <Vd>.<Ta>, <Vn>.<Tb>

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

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
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;

if acc then result = V[d];
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>;
    sum = (op1 + op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	23	22	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	0	Rn						Rd					
U										op																							

SADDLP <Vd>.<Ta>, <Vn>.<Tb>

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

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
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;

if acc then result = V[d];
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>;
    sum = (op1 + op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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		ftype		0		0		0		0		1		0		scale				Rn				Rd			
rmode																opcode																									

### 32-bit to half-precision (sf == 0 && ftype == 11) (FEAT\_FP16)

SCVTF <Hd>, <Wn>, #<fbits>

### 32-bit to single-precision (sf == 0 && ftype == 00)

SCVTF <Sd>, <Wn>, #<fbits>

### 32-bit to double-precision (sf == 0 && ftype == 01)

SCVTF <Dd>, <Wn>, #<fbits>

### 64-bit to half-precision (sf == 1 && ftype == 11) (FEAT\_FP16)

SCVTF <Hd>, <Xn>, #<fbits>

### 64-bit to single-precision (sf == 1 && ftype == 00)

SCVTF <Sd>, <Xn>, #<fbits>

### 64-bit to double-precision (sf == 1 && ftype == 01)

SCVTF <Dd>, <Xn>, #<fbits>

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPCConvOp op;
FPRounding rounding;
boolean unsigned;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = case opcode<2:1>:rmode of
  when '00 11' // FCVTZ
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPCConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR[]);
    unsigned = (opcode<0> == '1');
    op = FPCConvOp_CVT_ItoF(FPCR[]);
  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> 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".  
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".

## Operation

```

CheckFPAdvSIMDEnabled64();

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = integer fsize = if op == FPCnvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when FPCnvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, fracbits, unsigned, fpcr, rounding);
    X[d] = intval;
  when FPCnvOp_CVT_ItoF
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, FALSE, fpcr, rounding);
    V[d] = fltval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

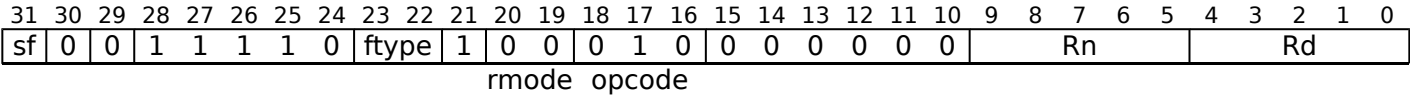
(new)

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 && ftype == 11)**  
(FEAT\_FP16)

SCVTF <Hd>, <Wn>

**32-bit to single-precision (sf == 0 && ftype == 00)**

SCVTF <Sd>, <Wn>

**32-bit to double-precision (sf == 0 && ftype == 01)**

SCVTF <Dd>, <Wn>

**64-bit to half-precision (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

SCVTF <Hd>, <Xn>

**64-bit to single-precision (sf == 1 && ftype == 00)**

SCVTF <Sd>, <Xn>

**64-bit to double-precision (sf == 1 && ftype == 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 ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = 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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(FPCR[]);
  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();

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = integer fsize = if op == FPCConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when FPCConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, fpcr, rounding);
    X[n];
  fltval = if merge then [d] = intval;
  when FPCConvOp_CVT_ItoF
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, FALSE, fpcr, rounding); (intval, 0, unsigned, fpcr, rounding)
V[d] = fltval;
  when FPCConvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPCConvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPCConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = fltval; [d] = intval;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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

(FEAT\_DotProd)

31	30	29	28	27	26	25	24	23	22	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	1	0	H	0					Rn				Rd	
U																															

SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]

```

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
if size != '10' then UNDEFINED;
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 third source and 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]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
[operand2, 4 * index + i, esize DIV 4]);
else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14.5341  
Copyright © 2010-2020 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 signed 8-bit elements in each 32-bit element of the first source register with the four signed 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.

### Vector (FEAT\_DotProd)

31	30	29	28	27	26	25	24	23	22	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																															

SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```

if !HaveDOTPEExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
if size != '10' then UNDEFINED;
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 third source and 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]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
[operand2, 4 * e + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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						

SHA1C <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

## 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); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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										Rd

SHA1M <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

## 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); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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				

SHA1P <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

## 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); {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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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						

SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

## 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 = 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## SHA256H2

SHA256 hash update (part 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	0	Rm					0	1	0	1	0	0	Rn					Rd				
P																															

SHA256H2 <Qd>, <Qn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;() then UNDEFINED;
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;
result =if part1 then
    result = SHA256hash(V[n],[d], V[d],[n], V[m], FALSE);[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SHA256H

SHA256 hash update (part 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	0	0	0	0	Rn				Rd						
P																															

SHA256H <Qd>, <Qn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;() then UNDEFINED;
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;
result =if part1 then
    result = SHA256hash(V[d], V[n], V[m], TRUE);[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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					

SHA256SU0 <Vd>.4S, <Vn>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA256Ext() then UNDEFINED;
```

### 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(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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						

SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

### 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.CheckFPAAdvSIMDEnabled();

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(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];
    [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 [FEAT\\_SHA512](#) is implemented.

### Advanced SIMD (FEAT\_SHA512)

31	30	29	28	27	26	25	24	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									Rd

SHA512H2 <Qd>, <Qn>, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
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(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(Y<63:0>,34) EOR ROR(Y<63:0>, 39);
(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>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 + W<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR(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>);(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 [FEAT\\_SHA512](#) is implemented.

### Advanced SIMD (FEAT\_SHA512)

31	30	29	28	27	26	25	24	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						

SHA512H <Qd>, <Qn>, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
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(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>);
(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(tmp, 18) EOR ROR(tmp, 41);
(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)

## 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 [FEAT\\_SHA512](#) is implemented.

### Advanced SIMD (FEAT\_SHA512)

31	30	29	28	27	26	25	24	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	Rn				Rd						

SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
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.

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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 [FEAT\\_SM3](#) is implemented.

### Advanced SIMD (FEAT\_SM3)

31	30	29	28	27	26	25	24	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

SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM3Ext() then UNDEFINED;
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):(Vm<127:96>,15):ROL(Vm<95:64>, 15):(Vm<95:64>,15):ROL(Vm<63:32>,15));
```

```
for i = 0 to 3
```

```
    if i == 3 then
    if i == 3 then
```

```
        result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
```

```
        (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(result<(32*i)+31:(32*i)>,15);
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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 [FEAT\\_SM3](#) is implemented.

### Advanced SIMD (FEAT\_SM3)

31	30	29	28	27	26	25	24	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					Rn				Rd

SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM3Ext() then UNDEFINED;
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) : (Vm<127:96>, 7) : ROL(Vm<95:64>, 7) : (Vm<95:64>, 7) : ROL(Vm<63:32>, 7) : (Vm<31:0>, 7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
(tmp<31:0>, 15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR (tmp2, 15) EOR ROL(tmp2, 23);
(tmp2, 23);
result<127:96> = result<127:96> EOR tmp2;
V[d] = result; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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 *FEAT\_SM3* is implemented.

### Advanced SIMD (FEAT\_SM3)

31	30	29	28	27	26	25	24	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				0	Ra				Rn				Rd							

SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S

```
if !HaveSM3Ext() then UNDEFINED;
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);
(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.

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14.5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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 *FEAT\_SM3* is implemented.

### Advanced SIMD (FEAT\_SM3)

31	30	29	28	27	26	25	24	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				

SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
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];
[Vm,i,32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
(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>;
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>;
(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 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 [FEAT\\_SM3](#) is implemented.

### Advanced SIMD (FEAT\_SM3)

31	30	29	28	27	26	25	24	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						

SM3TT1B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
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];
[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>;
(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>;
(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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 [FEAT\\_SM3](#) is implemented.

### Advanced SIMD (FEAT\_SM3)

31	30	29	28	27	26	25	24	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					

SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
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];
[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>;
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>;
(Vd<95:64>,19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR (TT2,9) EOR ROL(TT2, 17);(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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 *FEAT\_SM3* is implemented.

### Advanced SIMD (FEAT\_SM3)

31	30	29	28	27	26	25	24	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							

SM3TT2B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
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>;
[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>;
(Vd<95:64>,19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR (TT2,9) EOR ROL(TT2, 17); (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 *FEAT\_SM4* is implemented.

### Advanced SIMD (FEAT\_SM4)

31	30	29	28	27	26	25	24	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								

SM4E <Vd>.4S, <Vn>.4S

```
if !HaveSM4Ext() then UNDEFINED;
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(32) intval;
```

```
bits(8) sboxout;
```

```
bits(128) roundresult;
```

```
bits(32) roundkey;
```

```
roundresult = roundresult = V[d];
```

```
for index = 0 to 3
```

```
    roundkey = Elem[Vn, index, 32];
```

```
[Vn, index, 32];
```

```
    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;
```

```
    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;
```

```
    for i = 0 to 3
```

```
        Elem[intval, i, 8] = [intval, i, 8] = Sbox(Elem[intval, i, 8]);
```

```
[intval, i, 8]);
```

```
        intval = intval EOR ROL(intval, 2) EOR (intval, 2) EOR ROL(intval, 10) EOR (intval, 10) EOR ROL(intval, 18) EOR (intval, 18);
```

```
        intval = intval EOR roundresult<31:0>;
```

```
        (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## 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 *FEAT\_SM4* is implemented.

### Advanced SIMD (FEAT\_SM4)

31	30	29	28	27	26	25	24	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						

SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM4Ext() then UNDEFINED;
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;
```

```
roundresult = V[n];
for index = 0 to 3
    const = Elem[Vm, index, 32];
    [Vm, index, 32];
```

```
intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
```

```
for i = 0 to 3
    Elem[intval, i, 8] = [intval, i, 8] = Sbox(Elem[intval, i, 8]);
    [intval, i, 8];
```

```
intval = intval EOR ROL(intval, 13) EOR (intval, 13) EOR ROL(intval, 23);
intval = intval EOR roundresult<31:0>;
(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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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\\_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	0	1	0	H	0											
U										o2																					
																				Rn											
																				Rd											

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 = 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>;
    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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																									

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 << 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>;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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					Rn				Rd	
U										o2																					

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 = 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>;
    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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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																									

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 << 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>;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## SMMLA (vector)

Signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of signed 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. [ID\\_AA64ISAR0\\_EL1](#).I8MM indicates whether this instruction is supported.

### Vector

(FEAT\_I8MM)

31	30	29	28	27	26	25	24	23	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	1	0	0	Rm				1	0	1	0	0	1	Rn				Rd						
U										B																					

SMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

```
if !HaveInt8MatMulExt() then UNDEFINED;
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

### Assembler Symbols

- <Vd> Is the name of the SIMD&FP third source and 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

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];
V[d] = MatMulAdd(addend, operand1, operand2, FALSE, FALSE);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

SMULL{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 = 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  $\langle Ts \rangle$  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(idxsizesize) 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>; 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53+01:00

Copyright © 2010-2020 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																															

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 << 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>;[result, e, 2*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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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														

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 = 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														

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 = 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);
    (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);
    (accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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

31	30	29	28	27	26	25	24	23	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	1	0	0				Rn				Rd		

o1

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 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '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	0	1	0	0				Rn				Rd		

o1

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 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

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

### Assembler Symbols

- 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);
(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);
(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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									Rd	
																	o2														

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 = 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				Rn					Rd		
																	o2														

SQDMLSL{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 = 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);
    (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);
    (accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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

31	30	29	28	27	26	25	24	23	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	0	Rn				Rd						
																	o1														

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 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '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	0	Rn				Rd						
																	o1														

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 << 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);
    (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);
    (accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z

Copyright © 2010-2020 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		

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 = 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		

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 = 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 <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;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat) = SignedSat0(2 * element1 * element2, 2 * esize); (2 * element1 * element2, 2*esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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](#), [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	1	0	1	0	0				Rn				Rd		

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 << 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		

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 << 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); (2 * element1 * element2, 2*esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff 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.

31	30	29	28	27	26	25	24	23	22	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	1	Rn						Rd				
U																																

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 << 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>; [result, e, esize] = (element1 + element2 + 1)
V[d] = result;
```

(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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

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;

// .LD format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

## 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<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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																					

ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

## 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;

// .LD format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: ~~2020-12-16T16:20:14~~2020-12-16T14:53:41

Copyright © 2010-2020 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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

**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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

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

31	30	29	28	27	26	25	24	23	22	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	1	0	0	size	Rn				Rt						
L										opcode																					

ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### 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	1	0	0	size	Rn					Rt					
L										opcode																					

### 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;
boolean tag_checked = wback || n != 31;
```

## 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;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: ~~2020-12-16T16:5341~~2020-12-16T14:5341

Copyright © 2010-2020 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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

**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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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																					

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;
boolean tag_checked = wback || n != 31;
```

### 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											
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;
boolean tag_checked = wback || n != 31;
```

## 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;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

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, address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: ~~2020-12-16T16:20:14~~2020-12-16T14:53:41

Copyright © 2010-2020 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\\_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	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;
boolean tag_checked = wback || n != 31;
```

### 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;
boolean tag_checked = wback || n != 31;
```

## 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 init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
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 = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(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 HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
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, [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, [address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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>}]

```
// Empty.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);
if opc == '11' then UNDEFINED;
integer scale = 2 + (Rt2); AccType acctype = AccType_VECSTREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

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      UNDEFINED;
        when Constraint_NOP        EndOfInstruction(imm7, 64), scale);
boolean tag_checked = n != 31;();
```

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

data1 = case memop of
    when MemOp_STORE
        data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, [address + 0, dbytes, acctype] = data1; AccType_VECSTREAMMem] = data1; [address +

    when
MemOp_LOAD
    data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype];
    data2 = [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
        XAccType_VECSTREAMMem] = data2; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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		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>	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);
if opc == '11' then UNDEFINED;
integer scale = 2 + (Rt2); AccType acctype = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

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 UNDEFINED;
    when Constraint_NOP EndOfInstruction(imm7, 64), scale);
boolean tag_checked = wback || n != 31;();
```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

data1 = case memop of
    when MemOp_STORE
        data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, [address + 0, dbytes, acctype] = data1; AccType_VECMem = data1; [address + dbytes, dbytes, acctype] = data2;

    when
MemOp_LOAD
    data1 = Mem[address+dbytes, dbytes, [address + 0, dbytes, acctype]];
    data2 = [address + dbytes, dbytes, acctype];
    if rt_unknown then
        data1 = bits(datasize) UNKNOWN;
        data2 = bits(datasize) UNKNOWN;
        V[t] = data1;
        VAccType_VECMem = data2;
    [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 = 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)

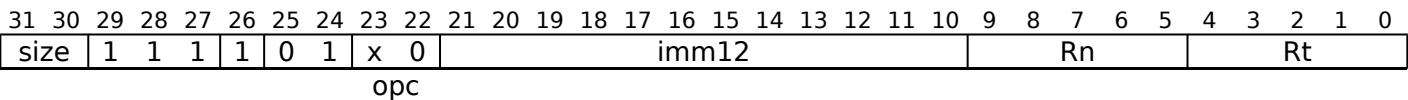
```
STR <Dt>, [<Xn|SP>, #<sim>]!
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<l>:size);
if scale > 4 then UNDEFINED;
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<l>:size);
if scale > 4 then UNDEFINED;
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.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	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.  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 <pimm>/2.  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.  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 <pimm>/16.

## 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;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype] = data;
when AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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 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;
boolean tag_checked = memop != MemOp_PREFETCH;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype] = data;

    when AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype]; AccType_VECV]; [t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        [] = address;
    else
        XVSP[t] = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 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<l>:size);
if scale > 4 then UNDEFINED;
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;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31); && (wback || n != 31);
```

## Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype]] = data;

    when AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, [address, datasize DIV 8, acctype]; AccType_VECV]; [t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        [] = address;
    else
        XVSP[t] = data; [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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## SUDOT (by element)

Dot product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination vector.

From Armv8.2, this is an OPTIONAL instruction. *ID\_AA64ISAR0\_EL1*.I8MM indicates whether this instruction is supported.

### Vector (FEAT\_I8MM)

31	30	29	28	27	26	25	24	23	22	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	L	M	Rm				1	1	1	1	H	0	Rn				Rd					
US																															

SUDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.*4B*[<index>]

```

if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;

```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and 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 immediate index of a quadruplet of four 8-bit elements in the range 0 to 3, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
  bits(32) res = Elem[operand3, e, 32];
  for b = 0 to 3
    integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
[operand1, 4 * e + b, 8], op1_unsigned);
    integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
[operand2, 4 * i + b, 8], op2_unsigned);
    res = res + element1 * element2;
    Elem[result, e, 32] = res;
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41  
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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;

// Create table from registers
for i = 0 to regs-1
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
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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;

// Create table from registers
for i = 0 to regs-1
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
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53.41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

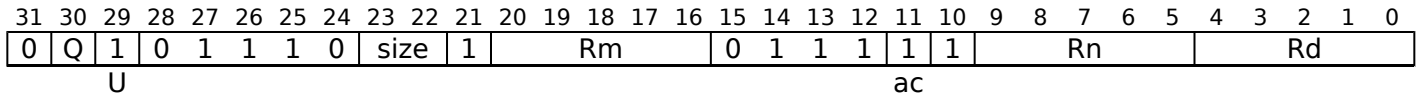
(new)



## 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.



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 << 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>; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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																						

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 << 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>; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

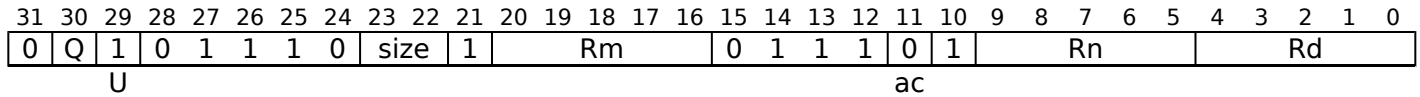
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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.



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 << 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>; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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																					

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 << 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>; (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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

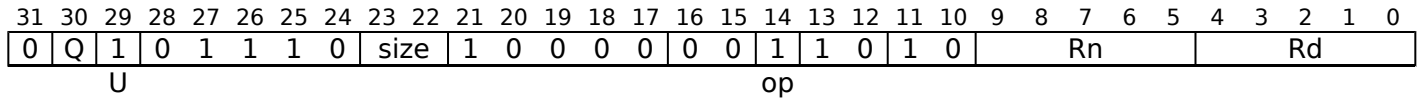
(old)	htmldiff from-	(new)
-------	----------------	-------



## 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.



UADALP <Vd>.<Ta>, <Vn>.<Tb>

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

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
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;

if acc then result = V[d];
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>;
    sum = (op1 + op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

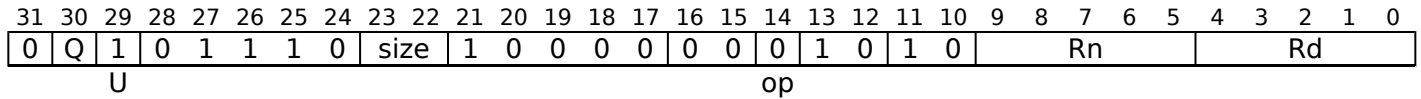
htmldiff from-

(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.



UADDLP <Vd>.<Ta>, <Vn>.<Tb>

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

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
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;

if acc then result = V[d];
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>;
    sum = (op1 + op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 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.

31	30	29	28	27	26	25	24	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		ftype		0		0		0		0		1		1		scale				Rn				Rd			
rmode																opcode																									

### 32-bit to half-precision (sf == 0 && ftype == 11) (FEAT\_FP16)

UCVTF <Hd>, <Wn>, #<fbits>

### 32-bit to single-precision (sf == 0 && ftype == 00)

UCVTF <Sd>, <Wn>, #<fbits>

### 32-bit to double-precision (sf == 0 && ftype == 01)

UCVTF <Dd>, <Wn>, #<fbits>

### 64-bit to half-precision (sf == 1 && ftype == 11) (FEAT\_FP16)

UCVTF <Hd>, <Xn>, #<fbits>

### 64-bit to single-precision (sf == 1 && ftype == 00)

UCVTF <Sd>, <Xn>, #<fbits>

### 64-bit to double-precision (sf == 1 && ftype == 01)

UCVTF <Dd>, <Xn>, #<fbits>

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

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPCConvOp op;
FPRounding rounding;
boolean unsigned;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = case opcode<2:1>:rmode of
  when '00 11' // FCVTZ
    rounding = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPCConvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR[]);
    unsigned = (opcode<0> == '1');
    op = FPCConvOp_CVT_ItoF(FPCR[]);
  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> 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".  
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".

## Operation

```

CheckFPAdvSIMDEnabled64();

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = integer fsize = if op == FPCnvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when FPCnvOp_CVT_FtoI
    fltval = V[n];
    intval = FPtoFixed(fltval, fracbits, unsigned, fpcr, rounding);
    X[d] = intval;
  when FPCnvOp_CVT_ItoF
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, TRUE, fpcr, rounding);
    V[d] = fltval;

```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

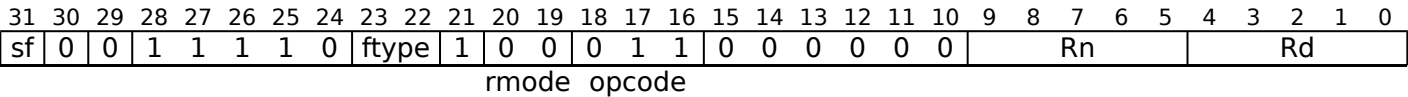
(new)

### 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 && ftype == 11)**  
(FEAT\_FP16)

UCVTF <Hd>, <Wn>

**32-bit to single-precision (sf == 0 && ftype == 00)**

UCVTF <Sd>, <Wn>

**32-bit to double-precision (sf == 0 && ftype == 01)**

UCVTF <Dd>, <Wn>

**64-bit to half-precision (sf == 1 && ftype == 11)**  
(FEAT\_FP16)

UCVTF <Hd>, <Xn>

**64-bit to single-precision (sf == 1 && ftype == 00)**

UCVTF <Sd>, <Xn>

**64-bit to double-precision (sf == 1 && ftype == 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 ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = 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 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 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 = FPRounding_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp_CVT_FtoI_JS(FPCR[]);
  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();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = integer fsize = if op == FPCnvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when FPCnvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, fpcr, rounding);
    X[n];
  fltval = if merge then [d] = intval;
  when FPCnvOp_CVT_ItoF
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, TRUE, fpcr, rounding); (intval, 0, unsigned, fpcr, rounding)
V[d] = fltval;
  when FPCnvOp_MOV_FtoI
    fltval = Vpart[n,part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPCnvOp_MOV_ItoF
    intval = X[n];
    fltval = intval<fsize-1:0>;
    Vpart[d,part] = fltval;
  when FPCnvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = fltval; [d] = intval;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## 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

(FEAT\_DotProd)

31	30	29	28	27	26	25	24	23	22	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	0	H	0							Rn			Rd

U

UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.[4B](#)[<index>]

```

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
if size != '10' then UNDEFINED;
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 third source and 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]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
[operand2, 4 * index + i, esize DIV 4]);
else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341  
Copyright © 2010-2020 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 unsigned 8-bit elements in each 32-bit element of the first source register with the four unsigned 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.

### Vector (FEAT\_DotProd)

31	30	29	28	27	26	25	24	23	22	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	1	0	1	Rn						Rd					
U																																	

UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
if size != '10' then UNDEFINED;
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 third source and 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]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
[operand2, 4 * e + i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
[operand1, 4 * e + i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
[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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:14:5341

Copyright © 2010-2020 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																					

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 = 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>;
    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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:20-12-16T14:5341

Copyright © 2010-2020 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																									

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 << 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>;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 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																					

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 = 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>;
    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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:20Z 2020-12-16T14:53:41

Copyright © 2010-2020 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																									

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 << 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>;
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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

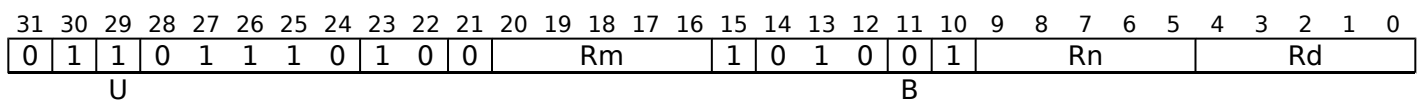
## UMMLA (vector)

Unsigned 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. [ID\\_AA64ISAR0\\_EL1](#).I8MM indicates whether this instruction is supported.

### Vector

(FEAT\_I8MM)



UMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

```
if !HaveInt8MatMulExt() then UNDEFINED;
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

### Assembler Symbols

- <Vd> Is the name of the SIMD&FP third source and 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

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];
V[d] = MatMulAdd(addend, operand1, operand2, TRUE, TRUE);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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 [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		1	0	1	0	H	0					Rn				Rd
U																															

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 = 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

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

**<Ts>** Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxsizesize) 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>; product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;

```

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 © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

Page 1044

## 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																															

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 << 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>;[result, e, 2*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 v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:2020-12-16T14:5341

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## 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.

31	30	29	28	27	26	25	24	23	22	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	1				Rn				Rd		

U

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 << 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>; [result, e, esize] = (element1 + element2 + 1)
V[d] = result;
```

(old)

htmldiff from-

(new)



## USDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. [ID\\_AA64ISAR0\\_EL1.I8MM](#) indicates whether this instruction is supported.

### Vector (FEAT\_I8MM)

31	30	29	28	27	26	25	24	23	22	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	L	M	Rm				1	1	1	1	H	0	Rn				Rd					
US																															

USDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm> .4B[<index>]

```
if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and 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 immediate index of a quadruplet of four 8-bit elements in the range 0 to 3, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
[operand1, 4 * e + b, 8], op1_unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
[operand2, 4 * i + b, 8], op2_unsigned);
        res = res + element1 * element2;
        Elem[result, e, 32] = res;
V[d] = result;
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:20:53Z 2020-12-16T14:53:41Z  
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

## USDOT (vector)

Dot Product vector form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. [ID\\_AA64ISAR0\\_EL1.I8MM](#) indicates whether this instruction is supported.

### Vector (FEAT\_I8MM)

31	30	29	28	27	26	25	24	23	22	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	0	Rm				1	0	0	1	1	1	Rn				Rd						

USDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and 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) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = UInt(Elem[operand1, 4*e+b, 8]);
        [operand1, 4 * e + b, 8]);
        integer element2 = SInt(Elem[operand2, 4*e+b, 8]);
        [operand2, 4 * e + b, 8]);
        res = res + element1 * element2;
        Elem[result, e, 32] = res;
V[d] = result;
```

(old)

htmldiff from-

(new)

## USMMLA (vector)

Unsigned and signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. [ID\\_AA64ISAR0\\_EL1](#).I8MM indicates whether this instruction is supported.

### Vector (FEAT\_I8MM)

31	30	29	28	27	26	25	24	23	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	1	0	0	Rm				1	0	1	0	1	1	Rn				Rd						
U										B																					

USMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

```
if !HaveInt8MatMulExt() then UNDEFINED;
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

### Assembler Symbols

- <Vd> Is the name of the SIMD&FP third source and 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

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];
V[d] = MatMulAdd(addend, operand1, operand2, TRUE, FALSE);
```

Internal version only: isa v32.13, AdvSIMD v29.05, pseudocode v2020-12, sve v2020-12 ; Build timestamp: 2020-12-16T16:53:41

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

## A64 -- SVE Instructions (alphabetic order)

ABS: Absolute value (predicated).

ADCLB: Add with carry long (bottom).

ADCLT: Add with carry long (top).

ADD (immediate): Add immediate (unpredicated).

ADD (vectors, predicated): Add vectors (predicated).

ADD (vectors, unpredicated): Add vectors (unpredicated).

ADDHNB: Add narrow high part (bottom).

ADDHNT: Add narrow high part (top).

ADDP: Add pairwise.

ADDPL: Add multiple of predicate register size to scalar register.

ADDVL: Add multiple of vector register size to scalar register.

ADR: Compute vector address.

AESD: AES single round decryption.

AESE: AES single round encryption.

AESIMC: AES inverse mix columns.

AESMC: AES mix columns.

AND (immediate): Bitwise AND with immediate (unpredicated).

AND (vectors, predicated): Bitwise AND vectors (predicated).

AND (vectors, unpredicated): Bitwise AND vectors (unpredicated).

AND, ANDS (predicates): Bitwise AND predicates.

ANDV: Bitwise AND reduction to scalar.

ASR (immediate, predicated): Arithmetic shift right by immediate (predicated).

ASR (immediate, unpredicated): Arithmetic shift right by immediate (unpredicated).

ASR (vectors): Arithmetic shift right by vector (predicated).

ASR (wide elements, predicated): Arithmetic shift right by 64-bit wide elements (predicated).

ASR (wide elements, unpredicated): Arithmetic shift right by 64-bit wide elements (unpredicated).

ASRD: Arithmetic shift right for divide by immediate (predicated).

ASRR: Reversed arithmetic shift right by vector (predicated).

BCAX: Bitwise clear and exclusive OR.

BDEP: Scatter lower bits into positions selected by bitmask.

BEXT: Gather lower bits from positions selected by bitmask.

BFCVT: Floating-point down convert to BFloat16 format (predicated).

BFCVTNT: Floating-point down convert and narrow to BFloat16 (top, predicated).

BFDOT (indexed): BFloat16 floating-point indexed dot product.

BFDOT (vectors): BFloat16 floating-point dot product.

BFMLALB (indexed): BFloat16 floating-point multiply-add long to single-precision (bottom, indexed).

BFMLALB (vectors): BFloat16 floating-point multiply-add long to single-precision (bottom).

BFMLALT (indexed): BFloat16 floating-point multiply-add long to single-precision (top, indexed).

BFMLALT (vectors): BFloat16 floating-point multiply-add long to single-precision (top).

BFMMLA: BFloat16 floating-point matrix multiply-accumulate.

BGRP: Group bits to right or left as selected by bitmask.

BIC (immediate): Bitwise clear bits using immediate (unpredicated): an alias of AND (immediate).

BIC (vectors, predicated): Bitwise clear vectors (predicated).

BIC (vectors, unpredicated): Bitwise clear vectors (unpredicated).

BIC, BICS (predicates): Bitwise clear predicates.

BRKA, BRKAS: Break after first true condition.

BRKB, BRKBS: Break before first true condition.

BRKN, BRKNS: Propagate break to next partition.

BRKPA, BRKPAS: Break after first true condition, propagating from previous partition.

BRKPB, BRKPBS: Break before first true condition, propagating from previous partition.

BSL: Bitwise select.

BSL1N: Bitwise select with first input inverted.

BSL2N: Bitwise select with second input inverted.

CADD: Complex integer add with rotate.

CDOT (indexed): Complex integer dot product (indexed).

CDOT (vectors): Complex integer dot product.

CLASTA (scalar): Conditionally extract element after last to general-purpose register.

CLASTA (SIMD&FP scalar): Conditionally extract element after last to SIMD&FP scalar register.

CLASTA (vectors): Conditionally extract element after last to vector register.

CLASTB (scalar): Conditionally extract last element to general-purpose register.

CLASTB (SIMD&FP scalar): Conditionally extract last element to SIMD&FP scalar register.

CLASTB (vectors): Conditionally extract last element to vector register.

CLS: Count leading sign bits (predicated).

CLZ: Count leading zero bits (predicated).

CMLA (indexed): Complex integer multiply-add with rotate (indexed).

CMLA (vectors): Complex integer multiply-add with rotate.

CMP<cc> (immediate): Compare vector to immediate.

CMP<cc> (vectors): Compare vectors.

CMP<cc> (wide elements): Compare vector to 64-bit wide elements.

CMPLE (vectors): Compare signed less than or equal to vector, setting the condition flags: an alias of CMP<cc> (vectors).

CMPLO (vectors): Compare unsigned lower than vector, setting the condition flags: an alias of CMP<cc> (vectors).

CMPLS (vectors): Compare unsigned lower or same as vector, setting the condition flags: an alias of CMP<cc> (vectors).

CMPLT (vectors): Compare signed less than vector, setting the condition flags: an alias of CMP<cc> (vectors).

CNOT: Logically invert boolean condition in vector (predicated).

CNT: Count non-zero bits (predicated).

CNTB, CNTD, CNTH, CNTW: Set scalar to multiple of predicate constraint element count.

CNTP: Set scalar to count of true predicate elements.

COMPACT: Shuffle active elements of vector to the right and fill with zero.

CPY (immediate, merging): Copy signed integer immediate to vector elements (merging).

CPY (immediate, zeroing): Copy signed integer immediate to vector elements (zeroing).

CPY (scalar): Copy general-purpose register to vector elements (predicated).

CPY (SIMD&FP scalar): Copy SIMD&FP scalar register to vector elements (predicated).

CTERMEQ, CTERMNE: Compare and terminate loop.

DECB, DECD, DECH, DECW (scalar): Decrement scalar by multiple of predicate constraint element count.

DECD, DECH, DECW (vector): Decrement vector by multiple of predicate constraint element count.

DECP (scalar): Decrement scalar by count of true predicate elements.

DECP (vector): Decrement vector by count of true predicate elements.

DUP (immediate): Broadcast signed immediate to vector elements (unpredicated).

DUP (indexed): Broadcast indexed element to vector (unpredicated).

DUP (scalar): Broadcast general-purpose register to vector elements (unpredicated).

DUPM: Broadcast logical bitmask immediate to vector (unpredicated).

EON: Bitwise exclusive OR with inverted immediate (unpredicated): an alias of EOR (immediate).

EOR (immediate): Bitwise exclusive OR with immediate (unpredicated).

EOR (vectors, predicated): Bitwise exclusive OR vectors (predicated).

EOR (vectors, unpredicated): Bitwise exclusive OR vectors (unpredicated).

EOR, EORS (predicates): Bitwise exclusive OR predicates.

EOR3: Bitwise exclusive OR of three vectors.

EORBT: Interleaving exclusive OR (bottom, top).

EORTB: Interleaving exclusive OR (top, bottom).

EORV: Bitwise exclusive OR reduction to scalar.

EXT: Extract vector from pair of vectors.

FABD: Floating-point absolute difference (predicated).

FABS: Floating-point absolute value (predicated).

FAC<cc>: Floating-point absolute compare vectors.



FACLE: Floating-point absolute compare less than or equal: an alias of FAC<cc>.

FACLT: Floating-point absolute compare less than: an alias of FAC<cc>.

FADD (immediate): Floating-point add immediate (predicated).

FADD (vectors, predicated): Floating-point add vector (predicated).

FADD (vectors, unpredicated): Floating-point add vector (unpredicated).

FADDA: Floating-point add strictly-ordered reduction, accumulating in scalar.

FADDP: Floating-point add pairwise.

FADDV: Floating-point add recursive reduction to scalar.

FCADD: Floating-point complex add with rotate (predicated).

FCM<cc> (vectors): Floating-point compare vectors.

FCM<cc> (zero): Floating-point compare vector with zero.

FCMLA (indexed): Floating-point complex multiply-add by indexed values with rotate.

FCMLA (vectors): Floating-point complex multiply-add with rotate (predicated).

FCMLE (vectors): Floating-point compare less than or equal to vector: an alias of FCM<cc> (vectors).

FCMLT (vectors): Floating-point compare less than vector: an alias of FCM<cc> (vectors).

FCPY: Copy 8-bit floating-point immediate to vector elements (predicated).

FCVT: Floating-point convert precision (predicated).

FCVTLT: Floating-point up convert long (top, predicated).

FCVTNT: Floating-point down convert and narrow (top, predicated).

FCVTX: Floating-point down convert, rounding to odd (predicated).

FCVTXNT: Floating-point down convert, rounding to odd (top, predicated).

FCVTZS: Floating-point convert to signed integer, rounding toward zero (predicated).

FCVTZU: Floating-point convert to unsigned integer, rounding toward zero (predicated).

FDIV: Floating-point divide by vector (predicated).

FDIVR: Floating-point reversed divide by vector (predicated).

FDUP: Broadcast 8-bit floating-point immediate to vector elements (unpredicated).

FEXPA: Floating-point exponential accelerator.

FLOGB: Floating-point base 2 logarithm as integer.

FMAD: Floating-point fused multiply-add vectors (predicated), writing multiplicand [ $Z_{dn} = Z_a + Z_{dn} * Z_m$ ].

FMAX (immediate): Floating-point maximum with immediate (predicated).

FMAX (vectors): Floating-point maximum (predicated).

FMAXNM (immediate): Floating-point maximum number with immediate (predicated).

FMAXNM (vectors): Floating-point maximum number (predicated).

FMAXNMP: Floating-point maximum number pairwise.

FMAXNMV: Floating-point maximum number recursive reduction to scalar.

FMAXP: Floating-point maximum pairwise.

FMAXV: Floating-point maximum recursive reduction to scalar.

FMIN (immediate): Floating-point minimum with immediate (predicated).

FMIN (vectors): Floating-point minimum (predicated).

FMINNM (immediate): Floating-point minimum number with immediate (predicated).

FMINNM (vectors): Floating-point minimum number (predicated).

FMINNMP: Floating-point minimum number pairwise.

FMINNMV: Floating-point minimum number recursive reduction to scalar.

FMINP: Floating-point minimum pairwise.

FMINV: Floating-point minimum recursive reduction to scalar.

FMLA (indexed): Floating-point fused multiply-add by indexed elements ( $Z_{da} = Z_{da} + Z_n * Z_m[\text{indexed}]$ ).

FMLA (vectors): Floating-point fused multiply-add vectors (predicated), writing addend [ $Z_{da} = Z_{da} + Z_n * Z_m$ ].

FMLALB (indexed): Half-precision floating-point multiply-add long to single-precision (bottom, indexed).

FMLALB (vectors): Half-precision floating-point multiply-add long to single-precision (bottom).

FMLALT (indexed): Half-precision floating-point multiply-add long to single-precision (top, indexed).

FMLALT (vectors): Half-precision floating-point multiply-add long to single-precision (top).

FMLS (indexed): Floating-point fused multiply-subtract by indexed elements ( $Z_{da} = Z_{da} + -Z_n * Z_m[\text{indexed}]$ ).

FMLS (vectors): Floating-point fused multiply-subtract vectors (predicated), writing addend [ $Z_{da} = Z_{da} + -Z_n * Z_m$ ].

FMLSBL (indexed): Half-precision floating-point multiply-subtract long from single-precision (bottom, indexed).

FMLSBL (vectors): Half-precision floating-point multiply-subtract long from single-precision (bottom).

FMLSBLT (indexed): Half-precision floating-point multiply-subtract long from single-precision (top, indexed).

FMLSBLT (vectors): Half-precision floating-point multiply-subtract long from single-precision (top).

FMMLA: Floating-point matrix multiply-accumulate.

FMOV (immediate, predicated): Move 8-bit floating-point immediate to vector elements (predicated): an alias of FCPY.

FMOV (immediate, unpredicated): Move 8-bit floating-point immediate to vector elements (unpredicated): an alias of FDUP.

FMOV (zero, predicated): Move floating-point +0.0 to vector elements (predicated): an alias of CPY (immediate, merging).

FMOV (zero, unpredicated): Move floating-point +0.0 to vector elements (unpredicated): an alias of DUP (immediate).

FMSB: Floating-point fused multiply-subtract vectors (predicated), writing multiplicand [ $Z_{dn} = Z_a + -Z_{dn} * Z_m$ ].

FMUL (immediate): Floating-point multiply by immediate (predicated).

FMUL (indexed): Floating-point multiply by indexed elements.

FMUL (vectors, predicated): Floating-point multiply vectors (predicated).

FMUL (vectors, unpredicated): Floating-point multiply vectors (unpredicated).

FMULX: Floating-point multiply-extended vectors (predicated).

FNEG: Floating-point negate (predicated).

FNMAV: Floating-point negated fused multiply-add vectors (predicated), writing multiplicand [ $Z_{dn} = -Z_a + -Z_{dn} * Z_m$ ].

FNMLA: Floating-point negated fused multiply-add vectors (predicated), writing addend [ $Z_{da} = -Z_{da} + -Z_n * Z_m$ ].

FNMLS: Floating-point negated fused multiply-subtract vectors (predicated), writing addend [ $Zda = -Zda + Z_n * Z_m$ ].

FNMSB: Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand [ $Zdn = -Za + Zdn * Zm$ ].

FRECPE: Floating-point reciprocal estimate (unpredicated).

FRECPS: Floating-point reciprocal step (unpredicated).

FRECPX: Floating-point reciprocal exponent (predicated).

FRINT< r >: Floating-point round to integral value (predicated).

FRSQRT: Floating-point reciprocal square root estimate (unpredicated).

FRSQRTS: Floating-point reciprocal square root step (unpredicated).

FSCALE: Floating-point adjust exponent by vector (predicated).

FSQRT: Floating-point square root (predicated).

FSUB (immediate): Floating-point subtract immediate (predicated).

FSUB (vectors, predicated): Floating-point subtract vectors (predicated).

FSUB (vectors, unpredicated): Floating-point subtract vectors (unpredicated).

FSUBR (immediate): Floating-point reversed subtract from immediate (predicated).

FSUBR (vectors): Floating-point reversed subtract vectors (predicated).

FTMAD: Floating-point trigonometric multiply-add coefficient.

FTSMUL: Floating-point trigonometric starting value.

FTSSEL: Floating-point trigonometric select coefficient.

HISTCNT: Count matching elements in vector.

HISTSEG: Count matching elements in vector segments.

INCB, INCD, INCH, INCW (scalar): Increment scalar by multiple of predicate constraint element count.

INCD, INCH, INCW (vector): Increment vector by multiple of predicate constraint element count.

INCP (scalar): Increment scalar by count of true predicate elements.

INCP (vector): Increment vector by count of true predicate elements.

INDEX (immediate, scalar): Create index starting from immediate and incremented by general-purpose register.

INDEX (immediates): Create index starting from and incremented by immediate.

INDEX (scalar, immediate): Create index starting from general-purpose register and incremented by immediate.

INDEX (scalars): Create index starting from and incremented by general-purpose register.

INSR (scalar): Insert general-purpose register in shifted vector.

INSR (SIMD&FP scalar): Insert SIMD&FP scalar register in shifted vector.

LASTA (scalar): Extract element after last to general-purpose register.

LASTA (SIMD&FP scalar): Extract element after last to SIMD&FP scalar register.

LASTB (scalar): Extract last element to general-purpose register.

LASTB (SIMD&FP scalar): Extract last element to SIMD&FP scalar register.

LD1B (scalar plus immediate): Contiguous load unsigned bytes to vector (immediate index).

LD1B (scalar plus scalar): Contiguous load unsigned bytes to vector (scalar index).

LD1B (scalar plus vector): Gather load unsigned bytes to vector (vector index).

LD1B (vector plus immediate): Gather load unsigned bytes to vector (immediate index).

LD1D (scalar plus immediate): Contiguous load doublewords to vector (immediate index).

LD1D (scalar plus scalar): Contiguous load doublewords to vector (scalar index).

LD1D (scalar plus vector): Gather load doublewords to vector (vector index).

LD1D (vector plus immediate): Gather load doublewords to vector (immediate index).

LD1H (scalar plus immediate): Contiguous load unsigned halfwords to vector (immediate index).

LD1H (scalar plus scalar): Contiguous load unsigned halfwords to vector (scalar index).

LD1H (scalar plus vector): Gather load unsigned halfwords to vector (vector index).

LD1H (vector plus immediate): Gather load unsigned halfwords to vector (immediate index).

LD1RB: Load and broadcast unsigned byte to vector.

LD1RD: Load and broadcast doubleword to vector.

LD1RH: Load and broadcast unsigned halfword to vector.

LD1ROB (scalar plus immediate): Contiguous load and replicate thirty-two bytes (immediate index).

LD1ROB (scalar plus scalar): Contiguous load and replicate thirty-two bytes (scalar index).

LD1ROD (scalar plus immediate): Contiguous load and replicate four doublewords (immediate index).

LD1ROD (scalar plus scalar): Contiguous load and replicate four doublewords (scalar index).

LD1ROH (scalar plus immediate): Contiguous load and replicate sixteen halfwords (immediate index).

LD1ROH (scalar plus scalar): Contiguous load and replicate sixteen halfwords (scalar index).

LD1ROW (scalar plus immediate): Contiguous load and replicate eight words (immediate index).

LD1ROW (scalar plus scalar): Contiguous load and replicate eight words (scalar index).

LD1RQB (scalar plus immediate): Contiguous load and replicate sixteen bytes (immediate index).

LD1RQB (scalar plus scalar): Contiguous load and replicate sixteen bytes (scalar index).

LD1RQD (scalar plus immediate): Contiguous load and replicate two doublewords (immediate index).

LD1RQD (scalar plus scalar): Contiguous load and replicate two doublewords (scalar index).

LD1RQH (scalar plus immediate): Contiguous load and replicate eight halfwords (immediate index).

LD1RQH (scalar plus scalar): Contiguous load and replicate eight halfwords (scalar index).

LD1RQW (scalar plus immediate): Contiguous load and replicate four words (immediate index).

LD1RQW (scalar plus scalar): Contiguous load and replicate four words (scalar index).

LD1RSB: Load and broadcast signed byte to vector.

LD1RSH: Load and broadcast signed halfword to vector.

LD1RSW: Load and broadcast signed word to vector.

LD1RW: Load and broadcast unsigned word to vector.

LD1SB (scalar plus immediate): Contiguous load signed bytes to vector (immediate index).

LD1SB (scalar plus scalar): Contiguous load signed bytes to vector (scalar index).

LD1SB (scalar plus vector): Gather load signed bytes to vector (vector index).

LD1SB (vector plus immediate): Gather load signed bytes to vector (immediate index).

LD1SH (scalar plus immediate): Contiguous load signed halfwords to vector (immediate index).

LD1SH (scalar plus scalar): Contiguous load signed halfwords to vector (scalar index).

LD1SH (scalar plus vector): Gather load signed halfwords to vector (vector index).

LD1SH (vector plus immediate): Gather load signed halfwords to vector (immediate index).

LD1SW (scalar plus immediate): Contiguous load signed words to vector (immediate index).

LD1SW (scalar plus scalar): Contiguous load signed words to vector (scalar index).

LD1SW (scalar plus vector): Gather load signed words to vector (vector index).

LD1SW (vector plus immediate): Gather load signed words to vector (immediate index).

LD1W (scalar plus immediate): Contiguous load unsigned words to vector (immediate index).

LD1W (scalar plus scalar): Contiguous load unsigned words to vector (scalar index).

LD1W (scalar plus vector): Gather load unsigned words to vector (vector index).

LD1W (vector plus immediate): Gather load unsigned words to vector (immediate index).

LD2B (scalar plus immediate): Contiguous load two-byte structures to two vectors (immediate index).

LD2B (scalar plus scalar): Contiguous load two-byte structures to two vectors (scalar index).

LD2D (scalar plus immediate): Contiguous load two-doubleword structures to two vectors (immediate index).

LD2D (scalar plus scalar): Contiguous load two-doubleword structures to two vectors (scalar index).

LD2H (scalar plus immediate): Contiguous load two-halfword structures to two vectors (immediate index).

LD2H (scalar plus scalar): Contiguous load two-halfword structures to two vectors (scalar index).

LD2W (scalar plus immediate): Contiguous load two-word structures to two vectors (immediate index).

LD2W (scalar plus scalar): Contiguous load two-word structures to two vectors (scalar index).

LD3B (scalar plus immediate): Contiguous load three-byte structures to three vectors (immediate index).

LD3B (scalar plus scalar): Contiguous load three-byte structures to three vectors (scalar index).

LD3D (scalar plus immediate): Contiguous load three-doubleword structures to three vectors (immediate index).

LD3D (scalar plus scalar): Contiguous load three-doubleword structures to three vectors (scalar index).

LD3H (scalar plus immediate): Contiguous load three-halfword structures to three vectors (immediate index).

LD3H (scalar plus scalar): Contiguous load three-halfword structures to three vectors (scalar index).

LD3W (scalar plus immediate): Contiguous load three-word structures to three vectors (immediate index).

LD3W (scalar plus scalar): Contiguous load three-word structures to three vectors (scalar index).

LD4B (scalar plus immediate): Contiguous load four-byte structures to four vectors (immediate index).

LD4B (scalar plus scalar): Contiguous load four-byte structures to four vectors (scalar index).

LD4D (scalar plus immediate): Contiguous load four-doubleword structures to four vectors (immediate index).

LD4D (scalar plus scalar): Contiguous load four-doubleword structures to four vectors (scalar index).

LD4H (scalar plus immediate): Contiguous load four-halfword structures to four vectors (immediate index).

LD4H (scalar plus scalar): Contiguous load four-halfword structures to four vectors (scalar index).

LD4W (scalar plus immediate): Contiguous load four-word structures to four vectors (immediate index).

LD4W (scalar plus scalar): Contiguous load four-word structures to four vectors (scalar index).

LDFF1B (scalar plus scalar): Contiguous load first-fault unsigned bytes to vector (scalar index).

LDFF1B (scalar plus vector): Gather load first-fault unsigned bytes to vector (vector index).

LDFF1B (vector plus immediate): Gather load first-fault unsigned bytes to vector (immediate index).

LDFF1D (scalar plus scalar): Contiguous load first-fault doublewords to vector (scalar index).

LDFF1D (scalar plus vector): Gather load first-fault doublewords to vector (vector index).

LDFF1D (vector plus immediate): Gather load first-fault doublewords to vector (immediate index).

LDFF1H (scalar plus scalar): Contiguous load first-fault unsigned halfwords to vector (scalar index).

LDFF1H (scalar plus vector): Gather load first-fault unsigned halfwords to vector (vector index).

LDFF1H (vector plus immediate): Gather load first-fault unsigned halfwords to vector (immediate index).

LDFF1SB (scalar plus scalar): Contiguous load first-fault signed bytes to vector (scalar index).

LDFF1SB (scalar plus vector): Gather load first-fault signed bytes to vector (vector index).

LDFF1SB (vector plus immediate): Gather load first-fault signed bytes to vector (immediate index).

LDFF1SH (scalar plus scalar): Contiguous load first-fault signed halfwords to vector (scalar index).

LDFF1SH (scalar plus vector): Gather load first-fault signed halfwords to vector (vector index).

LDFF1SH (vector plus immediate): Gather load first-fault signed halfwords to vector (immediate index).

LDFF1SW (scalar plus scalar): Contiguous load first-fault signed words to vector (scalar index).

LDFF1SW (scalar plus vector): Gather load first-fault signed words to vector (vector index).

LDFF1SW (vector plus immediate): Gather load first-fault signed words to vector (immediate index).

LDFF1W (scalar plus scalar): Contiguous load first-fault unsigned words to vector (scalar index).

LDFF1W (scalar plus vector): Gather load first-fault unsigned words to vector (vector index).

LDFF1W (vector plus immediate): Gather load first-fault unsigned words to vector (immediate index).

LDNF1B: Contiguous load non-fault unsigned bytes to vector (immediate index).

LDNF1D: Contiguous load non-fault doublewords to vector (immediate index).

LDNF1H: Contiguous load non-fault unsigned halfwords to vector (immediate index).

LDNF1SB: Contiguous load non-fault signed bytes to vector (immediate index).

LDNF1SH: Contiguous load non-fault signed halfwords to vector (immediate index).

LDNF1SW: Contiguous load non-fault signed words to vector (immediate index).

LDNF1W: Contiguous load non-fault unsigned words to vector (immediate index).

LDNT1B (scalar plus immediate): Contiguous load non-temporal bytes to vector (immediate index).

LDNT1B (scalar plus scalar): Contiguous load non-temporal bytes to vector (scalar index).

LDNT1B (vector plus scalar): Gather load non-temporal unsigned bytes.

LDNT1D (scalar plus immediate): Contiguous load non-temporal doublewords to vector (immediate index).

LDNT1D (scalar plus scalar): Contiguous load non-temporal doublewords to vector (scalar index).

LDNT1D (vector plus scalar): Gather load non-temporal unsigned doublewords.

LDNT1H (scalar plus immediate): Contiguous load non-temporal halfwords to vector (immediate index).

LDNT1H (scalar plus scalar): Contiguous load non-temporal halfwords to vector (scalar index).

LDNT1H (vector plus scalar): Gather load non-temporal unsigned halfwords.

LDNT1SB: Gather load non-temporal signed bytes.

LDNT1SH: Gather load non-temporal signed halfwords.

LDNT1SW: Gather load non-temporal signed words.

LDNT1W (scalar plus immediate): Contiguous load non-temporal words to vector (immediate index).

LDNT1W (scalar plus scalar): Contiguous load non-temporal words to vector (scalar index).

LDNT1W (vector plus scalar): Gather load non-temporal unsigned words.

LDR (predicate): Load predicate register.

LDR (vector): Load vector register.

LSL (immediate, predicated): Logical shift left by immediate (predicated).

LSL (immediate, unpredicated): Logical shift left by immediate (unpredicated).

LSL (vectors): Logical shift left by vector (predicated).

LSL (wide elements, predicated): Logical shift left by 64-bit wide elements (predicated).

LSL (wide elements, unpredicated): Logical shift left by 64-bit wide elements (unpredicated).

LSLR: Reversed logical shift left by vector (predicated).

LSR (immediate, predicated): Logical shift right by immediate (predicated).

LSR (immediate, unpredicated): Logical shift right by immediate (unpredicated).

LSR (vectors): Logical shift right by vector (predicated).

LSR (wide elements, predicated): Logical shift right by 64-bit wide elements (predicated).

LSR (wide elements, unpredicated): Logical shift right by 64-bit wide elements (unpredicated).

LSRR: Reversed logical shift right by vector (predicated).

MAD: Multiply-add vectors (predicated), writing multiplicand [ $Z_{dn} = Z_a + Z_{dn} * Z_m$ ].

MATCH: Detect any matching elements, setting the condition flags.

MLA (indexed): Multiply-add to accumulator (indexed).

MLA (vectors): Multiply-add vectors (predicated), writing addend [ $Z_{da} = Z_{da} + Z_n * Z_m$ ].

MLS (indexed): Multiply-subtract from accumulator (indexed).

MLS (vectors): Multiply-subtract vectors (predicated), writing addend [ $Z_{da} = Z_{da} - Z_n * Z_m$ ].

MOV (bitmask immediate): Move logical bitmask immediate to vector (unpredicated): an alias of DUPM.

MOV (immediate, predicated, merging): Move signed integer immediate to vector elements (merging): an alias of CPY (immediate, merging).

MOV (immediate, predicated, zeroing): Move signed integer immediate to vector elements (zeroing): an alias of CPY (immediate, zeroing).

MOV (immediate, unpredicated): Move signed immediate to vector elements (unpredicated): an alias of DUP (immediate).

MOV (predicate, predicated, merging): Move predicates (merging): an alias of SEL (predicates).

MOV (predicate, predicated, zeroing): Move predicates (zeroing): an alias of AND, ANDS (predicates).

MOV (predicate, unpredicated): Move predicate (unpredicated): an alias of ORR, ORRS (predicates).

MOV (scalar, predicated): Move general-purpose register to vector elements (predicated): an alias of CPY (scalar).

MOV (scalar, unpredicated): Move general-purpose register to vector elements (unpredicated): an alias of DUP (scalar).

MOV (SIMD&FP scalar, predicated): Move SIMD&FP scalar register to vector elements (predicated): an alias of CPY (SIMD&FP scalar).

MOV (SIMD&FP scalar, unpredicated): Move indexed element or SIMD&FP scalar to vector (unpredicated): an alias of DUP (indexed).

MOV (vector, predicated): Move vector elements (predicated): an alias of SEL (vectors).

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

MOVPRFX (predicated): Move prefix (predicated).

MOVPRFX (unpredicated): Move prefix (unpredicated).

MOVS (predicated): Move predicates (zeroing), setting the condition flags: an alias of AND, ANDS (predicates).

MOVS (unpredicated): Move predicate (unpredicated), setting the condition flags: an alias of ORR, ORRS (predicates).

MSB: Multiply-subtract vectors (predicated), writing multiplicand [ $Z_{dn} = Z_a - Z_{dn} * Z_m$ ].

MUL (immediate): Multiply by immediate (unpredicated).

MUL (indexed): Multiply (indexed).

MUL (vectors, predicated): Multiply vectors (predicated).

MUL (vectors, unpredicated): Multiply vectors (unpredicated).

NAND, NANDS: Bitwise NAND predicates.

NBSL: Bitwise inverted select.

NEG: Negate (predicated).

NMATCH: Detect no matching elements, setting the condition flags.

NOR, NORs: Bitwise NOR predicates.

NOT (predicate): Bitwise invert predicate: an alias of EOR, EORS (predicates).

NOT (vector): Bitwise invert vector (predicated).

NOTS: Bitwise invert predicate, setting the condition flags: an alias of EOR, EORS (predicates).

ORN (immediate): Bitwise inclusive OR with inverted immediate (unpredicated): an alias of ORR (immediate).

ORN, ORNS (predicates): Bitwise inclusive OR inverted predicate.

ORR (immediate): Bitwise inclusive OR with immediate (unpredicated).

ORR (vectors, predicated): Bitwise inclusive OR vectors (predicated).

ORR (vectors, unpredicated): Bitwise inclusive OR vectors (unpredicated).

ORR, ORRS (predicates): Bitwise inclusive OR predicate.

ORV: Bitwise inclusive OR reduction to scalar.

PFALSE: Set all predicate elements to false.

PFIRST: Set the first active predicate element to true.

PMUL: Polynomial multiply vectors (unpredicated).

PMULLB: Polynomial multiply long (bottom).

PMULLT: Polynomial multiply long (top).



PNEXT: Find next active predicate.

PRFB (scalar plus immediate): Contiguous prefetch bytes (immediate index).

PRFB (scalar plus scalar): Contiguous prefetch bytes (scalar index).

PRFB (scalar plus vector): Gather prefetch bytes (scalar plus vector).

PRFB (vector plus immediate): Gather prefetch bytes (vector plus immediate).

PRFD (scalar plus immediate): Contiguous prefetch doublewords (immediate index).

PRFD (scalar plus scalar): Contiguous prefetch doublewords (scalar index).

PRFD (scalar plus vector): Gather prefetch doublewords (scalar plus vector).

PRFD (vector plus immediate): Gather prefetch doublewords (vector plus immediate).

PRFH (scalar plus immediate): Contiguous prefetch halfwords (immediate index).

PRFH (scalar plus scalar): Contiguous prefetch halfwords (scalar index).

PRFH (scalar plus vector): Gather prefetch halfwords (scalar plus vector).

PRFH (vector plus immediate): Gather prefetch halfwords (vector plus immediate).

PRFW (scalar plus immediate): Contiguous prefetch words (immediate index).

PRFW (scalar plus scalar): Contiguous prefetch words (scalar index).

PRFW (scalar plus vector): Gather prefetch words (scalar plus vector).

PRFW (vector plus immediate): Gather prefetch words (vector plus immediate).

PTEST: Set condition flags for predicate.

PTRUE, PTRUES: Initialise predicate from named constraint.

PUNPKHI, PUNPKLO: Unpack and widen half of predicate.

RADDHNB: Rounding add narrow high part (bottom).

RADDHNT: Rounding add narrow high part (top).

RAX1: Bitwise rotate left by 1 and exclusive OR.

RBIT: Reverse bits (predicated).

RDDFFR (unpredicated): Read the first-fault register.

RDDFFR, RDDFRS (predicated): Return predicate of successfully loaded elements.

RDVL: Read multiple of vector register size to scalar register.

REV (predicate): Reverse all elements in a predicate.

REV (vector): Reverse all elements in a vector (unpredicated).

REVB, REVH, REVW: Reverse bytes / halfwords / words within elements (predicated).

RSHRNB: Rounding shift right narrow by immediate (bottom).

RSHRNT: Rounding shift right narrow by immediate (top).

RSUBHNB: Rounding subtract narrow high part (bottom).

RSUBHNT: Rounding subtract narrow high part (top).

SABA: Signed absolute difference and accumulate.

SABALB: Signed absolute difference and accumulate long (bottom).

SABALT: Signed absolute difference and accumulate long (top).

SABD: Signed absolute difference (predicated).

SABDLB: Signed absolute difference long (bottom).

SABDLT: Signed absolute difference long (top).

SADALP: Signed add and accumulate long pairwise.

SADDLB: Signed add long (bottom).

SADDLBT: Signed add long (bottom + top).

SADDLT: Signed add long (top).

SADDV: Signed add reduction to scalar.

SADDWB: Signed add wide (bottom).

SADDWT: Signed add wide (top).

SBCLB: Subtract with carry long (bottom).

SBCLT: Subtract with carry long (top).

SCVTF: Signed integer convert to floating-point (predicated).

SDIV: Signed divide (predicated).

SDIVR: Signed reversed divide (predicated).

SDOT (indexed): Signed integer indexed dot product.

SDOT (vectors): Signed integer dot product.

SEL (predicates): Conditionally select elements from two predicates.

SEL (vectors): Conditionally select elements from two vectors.

SETFFR: Initialise the first-fault register to all true.

SHADD: Signed halving addition.

SHRNB: Shift right narrow by immediate (bottom).

SHRNT: Shift right narrow by immediate (top).

SHSUB: Signed halving subtract.

SHSUBR: Signed halving subtract reversed vectors.

SLI: Shift left and insert (immediate).

SM4E: SM4 encryption and decryption.

SM4EKEY: SM4 key updates.

SMAX (immediate): Signed maximum with immediate (unpredicated).

SMAX (vectors): Signed maximum vectors (predicated).

SMAXP: Signed maximum pairwise.

SMA XV: Signed maximum reduction to scalar.

SMIN (immediate): Signed minimum with immediate (unpredicated).

SMIN (vectors): Signed minimum vectors (predicated).

SMINP: Signed minimum pairwise.

SMINV: Signed minimum reduction to scalar.

SMLALB (indexed): Signed multiply-add long to accumulator (bottom, indexed).

SMLALB (vectors): Signed multiply-add long to accumulator (bottom).

SMLALT (indexed): Signed multiply-add long to accumulator (top, indexed).

SMLALT (vectors): Signed multiply-add long to accumulator (top).

SMLSLB (indexed): Signed multiply-subtract long from accumulator (bottom, indexed).

SMLSLB (vectors): Signed multiply-subtract long from accumulator (bottom).

SMLSLT (indexed): Signed multiply-subtract long from accumulator (top, indexed).

SMLSLT (vectors): Signed multiply-subtract long from accumulator (top).

SMMLA: Signed integer matrix multiply-accumulate.

SMULH (predicated): Signed multiply returning high half (predicated).

SMULH (unpredicated): Signed multiply returning high half (unpredicated).

SMULLB (indexed): Signed multiply long (bottom, indexed).

SMULLB (vectors): Signed multiply long (bottom).

SMULLT (indexed): Signed multiply long (top, indexed).

SMULLT (vectors): Signed multiply long (top).

SPLICE: Splice two vectors under predicate control.

SQABS: Signed saturating absolute value.

SQADD (immediate): Signed saturating add immediate (unpredicated).

SQADD (vectors, predicated): Signed saturating addition (predicated).

SQADD (vectors, unpredicated): Signed saturating add vectors (unpredicated).

SQCADD: Saturating complex integer add with rotate.

SQDECB: Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count.

SQDECD (scalar): Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count.

SQDECD (vector): Signed saturating decrement vector by multiple of 64-bit predicate constraint element count.

SQDECH (scalar): Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count.

SQDECH (vector): Signed saturating decrement vector by multiple of 16-bit predicate constraint element count.

SQDECP (scalar): Signed saturating decrement scalar by count of true predicate elements.

SQDECP (vector): Signed saturating decrement vector by count of true predicate elements.

SQDECW (scalar): Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count.

SQDECW (vector): Signed saturating decrement vector by multiple of 32-bit predicate constraint element count.

SQDMLALB (indexed): Signed saturating doubling multiply-add long to accumulator (bottom, indexed).

SQDMLALB (vectors): Signed saturating doubling multiply-add long to accumulator (bottom).

SQDMLALBT: Signed saturating doubling multiply-add long to accumulator (bottom  $\times$  top).

SQDMLALT (indexed): Signed saturating doubling multiply-add long to accumulator (top, indexed).

SQDMLALT (vectors): Signed saturating doubling multiply-add long to accumulator (top).

SQDMLSLB (indexed): Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed).

SQDMLSLB (vectors): Signed saturating doubling multiply-subtract long from accumulator (bottom).

SQDMLSLBT: Signed saturating doubling multiply-subtract long from accumulator (bottom  $\times$  top).

SQDMLSLT (indexed): Signed saturating doubling multiply-subtract long from accumulator (top, indexed).

SQDMLSLT (vectors): Signed saturating doubling multiply-subtract long from accumulator (top).

SQDMULH (indexed): Signed saturating doubling multiply high (indexed).

SQDMULH (vectors): Signed saturating doubling multiply high (unpredicated).

SQDMULLB (indexed): Signed saturating doubling multiply long (bottom, indexed).

SQDMULLB (vectors): Signed saturating doubling multiply long (bottom).

SQDMULLT (indexed): Signed saturating doubling multiply long (top, indexed).

SQDMULLT (vectors): Signed saturating doubling multiply long (top).

SQINCB: Signed saturating increment scalar by multiple of 8-bit predicate constraint element count.

SQINCD (scalar): Signed saturating increment scalar by multiple of 64-bit predicate constraint element count.

SQINCD (vector): Signed saturating increment vector by multiple of 64-bit predicate constraint element count.

SQINCH (scalar): Signed saturating increment scalar by multiple of 16-bit predicate constraint element count.

SQINCH (vector): Signed saturating increment vector by multiple of 16-bit predicate constraint element count.

SQINCP (scalar): Signed saturating increment scalar by count of true predicate elements.

SQINCP (vector): Signed saturating increment vector by count of true predicate elements.

SQINCW (scalar): Signed saturating increment scalar by multiple of 32-bit predicate constraint element count.

SQINCW (vector): Signed saturating increment vector by multiple of 32-bit predicate constraint element count.

SQNEG: Signed saturating negate.

SQRDCMLAH (indexed): Saturating rounding doubling complex integer multiply-add high with rotate (indexed).

SQRDCMLAH (vectors): Saturating rounding doubling complex integer multiply-add high with rotate.

SQRDMLAH (indexed): Signed saturating rounding doubling multiply-add high to accumulator (indexed).

SQRDMLAH (vectors): Signed saturating rounding doubling multiply-add high to accumulator (unpredicated).

SQRDMLSH (indexed): Signed saturating rounding doubling multiply-subtract high from accumulator (indexed).

SQRDMLSH (vectors): Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated).

SQRDMULH (indexed): Signed saturating rounding doubling multiply high (indexed).

SQRDMULH (vectors): Signed saturating rounding doubling multiply high (unpredicated).

SQRSHL: Signed saturating rounding shift left by vector (predicated).

SQRSHLR: Signed saturating rounding shift left reversed vectors (predicated).

SQRSHRNB: Signed saturating rounding shift right narrow by immediate (bottom).

SQRSHRNT: Signed saturating rounding shift right narrow by immediate (top).

SQRSHRUNB: Signed saturating rounding shift right unsigned narrow by immediate (bottom).

SQRSHRUNT: Signed saturating rounding shift right unsigned narrow by immediate (top).

SQSHL (immediate): Signed saturating shift left by immediate.

SQSHL (vectors): Signed saturating shift left by vector (predicated).

SQSHLR: Signed saturating shift left reversed vectors (predicated).

SQSHLU: Signed saturating shift left unsigned by immediate.

SQSHRNB: Signed saturating shift right narrow by immediate (bottom).

SQSHRNT: Signed saturating shift right narrow by immediate (top).

SQSHRUNB: Signed saturating shift right unsigned narrow by immediate (bottom).

SQSHRUNT: Signed saturating shift right unsigned narrow by immediate (top).

SQSUB (immediate): Signed saturating subtract immediate (unpredicated).

SQSUB (vectors, predicated): Signed saturating subtraction (predicated).

SQSUB (vectors, unpredicated): Signed saturating subtract vectors (unpredicated).

SQSUBR: Signed saturating subtraction reversed vectors (predicated).

SQXTNB: Signed saturating extract narrow (bottom).

SQXTNT: Signed saturating extract narrow (top).

SQXTUNB: Signed saturating unsigned extract narrow (bottom).

SQXTUNT: Signed saturating unsigned extract narrow (top).

SRHADD: Signed rounding halving addition.

SRI: Shift right and insert (immediate).

SRSHL: Signed rounding shift left by vector (predicated).

SRSHLR: Signed rounding shift left reversed vectors (predicated).

SRSHR: Signed rounding shift right by immediate.

RSRA: Signed rounding shift right and accumulate (immediate).

SSHLLB: Signed shift left long by immediate (bottom).

SSHLLT: Signed shift left long by immediate (top).

SSRA: Signed shift right and accumulate (immediate).

SSUBLB: Signed subtract long (bottom).

SSUBLBT: Signed subtract long (bottom - top).

SSUBLT: Signed subtract long (top).

SSUBLTB: Signed subtract long (top - bottom).

SSUBWB: Signed subtract wide (bottom).

SSUBWT: Signed subtract wide (top).

ST1B (scalar plus immediate): Contiguous store bytes from vector (immediate index).

ST1B (scalar plus scalar): Contiguous store bytes from vector (scalar index).

ST1B (scalar plus vector): Scatter store bytes from a vector (vector index).

ST1B (vector plus immediate): Scatter store bytes from a vector (immediate index).

ST1D (scalar plus immediate): Contiguous store doublewords from vector (immediate index).

ST1D (scalar plus scalar): Contiguous store doublewords from vector (scalar index).

ST1D (scalar plus vector): Scatter store doublewords from a vector (vector index).

ST1D (vector plus immediate): Scatter store doublewords from a vector (immediate index).

ST1H (scalar plus immediate): Contiguous store halfwords from vector (immediate index).

ST1H (scalar plus scalar): Contiguous store halfwords from vector (scalar index).

ST1H (scalar plus vector): Scatter store halfwords from a vector (vector index).

ST1H (vector plus immediate): Scatter store halfwords from a vector (immediate index).

ST1W (scalar plus immediate): Contiguous store words from vector (immediate index).

ST1W (scalar plus scalar): Contiguous store words from vector (scalar index).

ST1W (scalar plus vector): Scatter store words from a vector (vector index).

ST1W (vector plus immediate): Scatter store words from a vector (immediate index).

ST2B (scalar plus immediate): Contiguous store two-byte structures from two vectors (immediate index).

ST2B (scalar plus scalar): Contiguous store two-byte structures from two vectors (scalar index).

ST2D (scalar plus immediate): Contiguous store two-doubleword structures from two vectors (immediate index).

ST2D (scalar plus scalar): Contiguous store two-doubleword structures from two vectors (scalar index).

ST2H (scalar plus immediate): Contiguous store two-halfword structures from two vectors (immediate index).

ST2H (scalar plus scalar): Contiguous store two-halfword structures from two vectors (scalar index).

ST2W (scalar plus immediate): Contiguous store two-word structures from two vectors (immediate index).

ST2W (scalar plus scalar): Contiguous store two-word structures from two vectors (scalar index).

ST3B (scalar plus immediate): Contiguous store three-byte structures from three vectors (immediate index).

ST3B (scalar plus scalar): Contiguous store three-byte structures from three vectors (scalar index).

ST3D (scalar plus immediate): Contiguous store three-doubleword structures from three vectors (immediate index).

ST3D (scalar plus scalar): Contiguous store three-doubleword structures from three vectors (scalar index).

ST3H (scalar plus immediate): Contiguous store three-halfword structures from three vectors (immediate index).

ST3H (scalar plus scalar): Contiguous store three-halfword structures from three vectors (scalar index).

ST3W (scalar plus immediate): Contiguous store three-word structures from three vectors (immediate index).

ST3W (scalar plus scalar): Contiguous store three-word structures from three vectors (scalar index).

ST4B (scalar plus immediate): Contiguous store four-byte structures from four vectors (immediate index).

ST4B (scalar plus scalar): Contiguous store four-byte structures from four vectors (scalar index).

ST4D (scalar plus immediate): Contiguous store four-doubleword structures from four vectors (immediate index).

ST4D (scalar plus scalar): Contiguous store four-doubleword structures from four vectors (scalar index).

ST4H (scalar plus immediate): Contiguous store four-halfword structures from four vectors (immediate index).

ST4H (scalar plus scalar): Contiguous store four-halfword structures from four vectors (scalar index).

ST4W (scalar plus immediate): Contiguous store four-word structures from four vectors (immediate index).

ST4W (scalar plus scalar): Contiguous store four-word structures from four vectors (scalar index).

STNT1B (scalar plus immediate): Contiguous store non-temporal bytes from vector (immediate index).

STNT1B (scalar plus scalar): Contiguous store non-temporal bytes from vector (scalar index).

STNT1B (vector plus scalar): Scatter store non-temporal bytes.

STNT1D (scalar plus immediate): Contiguous store non-temporal doublewords from vector (immediate index).

STNT1D (scalar plus scalar): Contiguous store non-temporal doublewords from vector (scalar index).

STNT1D (vector plus scalar): Scatter store non-temporal doublewords.

STNT1H (scalar plus immediate): Contiguous store non-temporal halfwords from vector (immediate index).

STNT1H (scalar plus scalar): Contiguous store non-temporal halfwords from vector (scalar index).

STNT1H (vector plus scalar): Scatter store non-temporal halfwords.

STNT1W (scalar plus immediate): Contiguous store non-temporal words from vector (immediate index).

STNT1W (scalar plus scalar): Contiguous store non-temporal words from vector (scalar index).

STNT1W (vector plus scalar): Scatter store non-temporal words.

STR (predicate): Store predicate register.

STR (vector): Store vector register.

SUB (immediate): Subtract immediate (unpredicated).

SUB (vectors, predicated): Subtract vectors (predicated).

SUB (vectors, unpredicated): Subtract vectors (unpredicated).

SUBHNB: Subtract narrow high part (bottom).

SUBHNT: Subtract narrow high part (top).

SUBR (immediate): Reversed subtract from immediate (unpredicated).

SUBR (vectors): Reversed subtract vectors (predicated).

SUDOT: Signed by unsigned integer indexed dot product.

SUNPKHI, SUNPKLO: Signed unpack and extend half of vector.

SUQADD: Signed saturating addition of unsigned value.

SXTB, SXTH, SXTW: Signed byte / halfword / word extend (predicated).

TBL: Programmable table lookup in one or two vector table (zeroing).

TBX: Programmable table lookup in single vector table (merging).

TRN1, TRN2 (predicates): Interleave even or odd elements from two predicates.

TRN1, TRN2 (vectors): Interleave even or odd elements from two vectors.

UABA: Unsigned absolute difference and accumulate.

UABALB: Unsigned absolute difference and accumulate long (bottom).

UABALT: Unsigned absolute difference and accumulate long (top).

UABD: Unsigned absolute difference (predicated).

UABDLB: Unsigned absolute difference long (bottom).

UABDLT: Unsigned absolute difference long (top).

UADALP: Unsigned add and accumulate long pairwise.

UADDLB: Unsigned add long (bottom).

UADDLT: Unsigned add long (top).

UADDV: Unsigned add reduction to scalar.

UADDWB: Unsigned add wide (bottom).

UADDWT: Unsigned add wide (top).

UCVTF: Unsigned integer convert to floating-point (predicated).

UDIV: Unsigned divide (predicated).

UDIVR: Unsigned reversed divide (predicated).

UDOT (indexed): Unsigned integer indexed dot product.

UDOT (vectors): Unsigned integer dot product.

UHADD: Unsigned halving addition.

UHSUB: Unsigned halving subtract.

UHSUBR: Unsigned halving subtract reversed vectors.

UMAX (immediate): Unsigned maximum with immediate (unpredicated).

UMAX (vectors): Unsigned maximum vectors (predicated).

UMAXP: Unsigned maximum pairwise.

UMAXV: Unsigned maximum reduction to scalar.

UMIN (immediate): Unsigned minimum with immediate (unpredicated).

UMIN (vectors): Unsigned minimum vectors (predicated).

UMINP: Unsigned minimum pairwise.

UMINV: Unsigned minimum reduction to scalar.

UMLALB (indexed): Unsigned multiply-add long to accumulator (bottom, indexed).

UMLALB (vectors): Unsigned multiply-add long to accumulator (bottom).

UMLALT (indexed): Unsigned multiply-add long to accumulator (top, indexed).

UMLALT (vectors): Unsigned multiply-add long to accumulator (top).

UMLSLB (indexed): Unsigned multiply-subtract long from accumulator (bottom, indexed).

UMLSLB (vectors): Unsigned multiply-subtract long from accumulator (bottom).

UMLSLT (indexed): Unsigned multiply-subtract long from accumulator (top, indexed).

UMLSLT (vectors): Unsigned multiply-subtract long from accumulator (top).

UMMLA: Unsigned integer matrix multiply-accumulate.

UMULH (predicated): Unsigned multiply returning high half (predicated).

UMULH (unpredicated): Unsigned multiply returning high half (unpredicated).

UMULLB (indexed): Unsigned multiply long (bottom, indexed).

UMULLB (vectors): Unsigned multiply long (bottom).

UMULLT (indexed): Unsigned multiply long (top, indexed).

UMULLT (vectors): Unsigned multiply long (top).

UQADD (immediate): Unsigned saturating add immediate (unpredicated).

UQADD (vectors, predicated): Unsigned saturating addition (predicated).



UQADD (vectors, unpredicated): Unsigned saturating add vectors (unpredicated).

UQDECB: Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count.

UQDECD (scalar): Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count.

UQDECD (vector): Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count.

UQDECH (scalar): Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count.

UQDECH (vector): Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count.

UQDECP (scalar): Unsigned saturating decrement scalar by count of true predicate elements.

UQDECP (vector): Unsigned saturating decrement vector by count of true predicate elements.

UQDECW (scalar): Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count.

UQDECW (vector): Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count.

UQINCB: Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count.

UQINCD (scalar): Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count.

UQINCD (vector): Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count.

UQINCH (scalar): Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count.

UQINCH (vector): Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count.

UQINCP (scalar): Unsigned saturating increment scalar by count of true predicate elements.

UQINCP (vector): Unsigned saturating increment vector by count of true predicate elements.

UQINCW (scalar): Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count.

UQINCW (vector): Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count.

UQRSHL: Unsigned saturating rounding shift left by vector (predicated).

UQRSHLR: Unsigned saturating rounding shift left reversed vectors (predicated).

UQRSHRNB: Unsigned saturating rounding shift right narrow by immediate (bottom).

UQRSHRNT: Unsigned saturating rounding shift right narrow by immediate (top).

UQSHL (immediate): Unsigned saturating shift left by immediate.

UQSHL (vectors): Unsigned saturating shift left by vector (predicated).

UQSHLR: Unsigned saturating shift left reversed vectors (predicated).

UQSHRNB: Unsigned saturating shift right narrow by immediate (bottom).

UQSHRNT: Unsigned saturating shift right narrow by immediate (top).

UQSUB (immediate): Unsigned saturating subtract immediate (unpredicated).

UQSUB (vectors, predicated): Unsigned saturating subtraction (predicated).

UQSUB (vectors, unpredicated): Unsigned saturating subtract vectors (unpredicated).

UQSUBR: Unsigned saturating subtraction reversed vectors (predicated).

UQXTNB: Unsigned saturating extract narrow (bottom).

UQXTNT: Unsigned saturating extract narrow (top).

URECPE: Unsigned reciprocal estimate (predicated).

URHADD: Unsigned rounding halving addition.

URSHL: Unsigned rounding shift left by vector (predicated).

URSHLR: Unsigned rounding shift left reversed vectors (predicated).

URSHR: Unsigned rounding shift right by immediate.

URSQRTE: Unsigned reciprocal square root estimate (predicated).

URSRA: Unsigned rounding shift right and accumulate (immediate).

USDOT (indexed): Unsigned by signed integer indexed dot product.

USDOT (vectors): Unsigned by signed integer dot product.

USHLLB: Unsigned shift left long by immediate (bottom).

USHLLT: Unsigned shift left long by immediate (top).

USMMLA: Unsigned by signed integer matrix multiply-accumulate.

USQADD: Unsigned saturating addition of signed value.

USRA: Unsigned shift right and accumulate (immediate).

USUBLB: Unsigned subtract long (bottom).

USUBLT: Unsigned subtract long (top).

USUBWB: Unsigned subtract wide (bottom).

USUBWT: Unsigned subtract wide (top).

UUNPKHI, UUNPKLO: Unsigned unpack and extend half of vector.

UXTB, UXTH, UXTW: Unsigned byte / halfword / word extend (predicated).

UZP1, UZP2 (predicates): Concatenate even or odd elements from two predicates.

UZP1, UZP2 (vectors): Concatenate even or odd elements from two vectors.

WHILEGE: While decrementing signed scalar greater than or equal to scalar.

WHILEGT: While decrementing signed scalar greater than scalar.

WHILEHI: While decrementing unsigned scalar higher than scalar.

WHILEHS: While decrementing unsigned scalar higher or same as scalar.

WHILELE: While incrementing signed scalar less than or equal to scalar.

WHILELO: While incrementing unsigned scalar lower than scalar.

WHILELS: While incrementing unsigned scalar lower or same as scalar.

WHILELT: While incrementing signed scalar less than scalar.

WHILERW: While free of read-after-write conflicts.

WHILEWR: While free of write-after-read/write conflicts.

WRFFR: Write the first-fault register.

XAR: Bitwise exclusive OR and rotate right by immediate.

ZIP1, ZIP2 (predicates): Interleave elements from two half predicates.

ZIP1, ZIP2 (vectors): Interleave elements from two half vectors.

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