

(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 © 2021 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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21-03-30T21:41:22

Copyright © 2010-2021 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, CASAH, 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: an alias of DSB.

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: an alias of DSB.

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 ELO 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, LDSETHA, 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.

TLBI: TLB Invalidate operation: an alias of SYS.

TSB CSYNC: Trace Synchronization Barrier.

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

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

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

UBFM: Unsigned Bitfield Move.

UBFX: Unsigned Bitfield Extract: an alias of UBFM.

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, XPACI, XPACLRI: Strip Pointer Authentication Code.

YIELD: YIELD.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

(old)

htmldiff from-

(new)

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);
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

```
boolean branch_conditional = TRUE;
if ConditionHolds(condition) then(condition) then
    boolean branch_conditional = TRUE;
    BranchTo(PC[] + offset, BranchType_DIR, branch_conditional);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

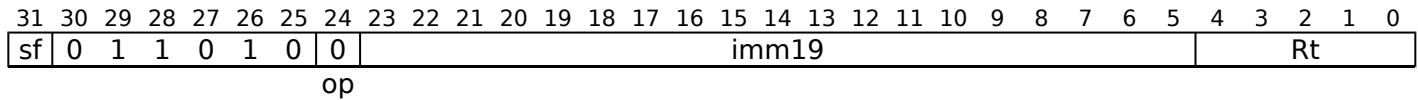
(old)

htmldiff from-

(new)

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];
boolean branch_conditional = TRUE;
if IsZero(operand1) == iszero then (operand1) == iszero then
    boolean branch_conditional = TRUE;
    BranchTo(PC[] + offset, BranchType_DIR, branch_conditional);
```

Internal version only: isa v32.21 v32.15, AdvSIMD v29.05, pseudocode v2021-06_xml v2021-03, sve v2021-06_rc2b v2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z 2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see [Data Synchronization Barrier](#).

A DSB instruction with the nXS qualifier is complete when the subset of these memory accesses with the XS attribute set to 0 are complete. It does not require that memory accesses with the XS attribute set to 1 are complete.

This instruction is used by the aliases [PSSBB](#), and [SSBB](#).

It has encodings from 2 classes: [Memory barrier](#) and [Memory nXS barrier](#)

Memory barrier

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

DSB <option>|<imm>

```
boolean nXS = FALSE;
```

```
case CRm of
  when '0000' alias = case CRm<3:2> of
  when '00' domain = DSBAlias_SSBB;
  when '0100' alias = DSBAlias_PSSBB;
  otherwise alias = DSBAlias_DSB;

case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;
case CRm<1:0> of
  when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;
```

Memory nXS barrier (FEAT_XS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	imm2	1	0	0	0	0	1	1	1	1	1	1

DSB <option>nXS|<imm>

```
if !HaveFeatXS() then UNDEFINED;
MBReqTypes types = MBReqTypes_All;
boolean nXS = TRUE; boolean nXS = TRUE;

case imm2 of
  when '00' domain =
DSBAlias alias = DSBAlias_DSB;

case imm2 of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;
```


Assembler Symbols

<option>	For the memory barrier variant: specifies the limitation on the barrier operation. Values are:
SY	Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.
ST	Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.
LD	Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.
ISH	Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.
ISHST	Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.
ISHLD	Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.
NSH	Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.
NSHST	Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.
NSHLD	Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.
OSH	Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.
OSHST	Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.
OSHLD	Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.
All other encodings of CRm, other than the values 0b0000 and 0b0100, that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB) .	
The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.	
	For the memory nXS barrier variant: specifies the limitation on the barrier operation. Values are:
SY	Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm<3:2> = 0b11.

- ISH**
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b10.
- NSH**
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b01.
- OSH**
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b00.

<imm> For the memory barrier variant: is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

For the memory nXS barrier variant: is a 5-bit unsigned immediate, encoded in "imm2":

imm2	<imm>
00	16
01	20
10	24
11	28

Alias Conditions

Alias	Is preferred when
PSSBB	CRm == '0100'
SSBB	CRm == '0000'

Operation

```

case alias of
  when if !nXS && DSBAlias_SSBBSpeculativeStoreBypassBarrierToVA();
  when DSBAlias_PSSBBSpeculativeStoreBypassBarrierToPA();
  when DSBAlias_DSB
    if !nXS && HaveFeatXS() && HaveFeatHCX() then
      nXS = PSTATE.EL IN {EL0, EL1} && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1';
      DataSynchronizationBarrier(domain, types, nXS);
    otherwise
      Unreachable();(domain, types, nXS);

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+00:002021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

HVC

Hypervisor Call causes an exception to EL2. ~~Software~~ ~~Non-secure software~~ executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- ~~When~~ ~~At~~ EL3 is implemented and ~~EL0~~, SCR_EL3.HCE is set to 0.
- When EL3 is not implemented and HCR_EL2.HCD is set to 1.
- When EL2 is not implemented.
- At EL1 if EL2 is not enabled in the current Security state.
- ~~At EL0~~. ~~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>

```
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(imm);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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>{, #<sim>}]
```

64-bit (size == 11)

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

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}]
```

```
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 = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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>}]
```

```
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 = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>**}]

```
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);
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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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. For information on single-copy atomicity and alignment requirements, see [Requirements for single-copy atomicity](#) and [Alignment of data accesses](#). The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics, as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

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

32-bit (sz == 0)

```
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

64-bit (sz == 1)

```
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
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[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) 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);

        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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
            else
                data = Mem[address, dbytes, acctype];
                X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

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

32-bit (size == 10)

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

64-bit (size == 11)

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC 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 = TRUE;    // address is UNKNOWN
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(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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                X[t] = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
            else
                data = Mem[address, dbytes, acctype];
                X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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);
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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
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(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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                X[t] = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
            else
                data = Mem[address, dbytes, acctype];
                X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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);
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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
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(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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
            else
                data = Mem[address, dbytes, acctype];
                X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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);
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      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) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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);
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);
bits(64) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:21+00:002021-03-30T21:41:22+00:00

Copyright © 2010-2021 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>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	1	0	imm9									1	1	Rn				Rt					
size										opc																					

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

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
0		0		1		1		1		0		0		1		0		1		imm12												Rn				Rt			
size										opc																													

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

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

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRH \(immediate\)](#).

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <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);
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      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) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

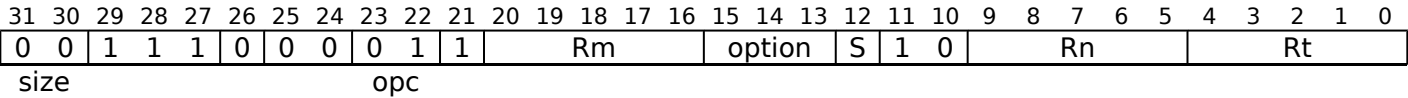
(old)

htmldiff from-

(new)

LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



Extended register (option != 011)

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

Shifted register (option == 011)

LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
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, shift);
bits(64) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:21+00:002021-03-30T21:41:22+00:00

Copyright © 2010-2021 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;
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;
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;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRH \(immediate\)](#).

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}}]

```
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.
- <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);
bits(64) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;
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;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
0		0		1		1		1		0		0		1		1		x		imm12											Rn				Rt			
size										opc																												

32-bit (opc == 11)

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

64-bit (opc == 10)

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

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

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRSB \(immediate\)](#).

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa ~~v32.21~~~~v32.15~~, AdvSIMD v29.05, pseudocode ~~v2021-06_xml~~~~v2021-03~~, sve ~~v2021-06_rc2~~~~v2021-03_rc2~~; Build timestamp: ~~2021-06-28T16:41:22~~~~2021-03-30T21:41:22~~

Copyright © 2010-2021 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>}]

```
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 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, shift);
bits(64) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;
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;
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;  
integer scale = UInt(size);  
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRSH \(immediate\)](#).

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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										S	1	0									
size								opc				Rm				option				Rn				Rt							

32-bit (opc == 11)

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

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;           // sub-word index
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> 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);
bits(64) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:21+00:002021-03-30T21:41:22+00:00

Copyright © 2010-2021 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;
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;
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;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRSW \(immediate\)](#).

Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}}]

```
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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:21+00:002021-03-30T21:41:22+00:00

Copyright © 2010-2021 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;

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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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>}]

```
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;

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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}]

```
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;

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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}]
```

```
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 = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}]
```

```
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 = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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>{, #<simm>}]

```
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);

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 = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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);
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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	1	0	imm9									0	0	Rn				Rt					
size											opc																				

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

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	1	0	imm9									0	0	Rn				Rt					
size								opc																							

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

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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>{, #<simm>}]

```
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);
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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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. For information on single-copy atomicity and alignment requirements, see [Requirements for single-copy atomicity](#) and [Alignment of data accesses](#). 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	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)	0	Rt2					Rn					Rt				
								L			Rs					o0															

32-bit (sz == 0)

LDXP <Wt1>, <Wt2>, [<Xn|SP>{, #0}]

64-bit (sz == 1)

LDXP <Xt1>, <Xt2>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_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        rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
rn_unknown = FALSE;    // address is original base
```

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

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) 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);

        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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
                else
                    data = Mem[address, dbytes, acctype];
                    X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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 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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
rn_unknown = FALSE;    // address is original base
```


Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.


```

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

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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
                else
                    data = Mem[address, dbytes, acctype];
                    X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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);
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 = TRUE;    // address is UNKNOWN
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(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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                X[t] = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
            else
                data = Mem[address, dbytes, acctype];
                X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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);
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 = TRUE;    // address is UNKNOWN
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(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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
                else
                    data = Mem[address, dbytes, acctype];
                    X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}]

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

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

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_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) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}}]

```
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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:21-03-30T21:4122

Copyright © 2010-2021 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>}]

```
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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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);
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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	0	1	1	0	0	1	0	0	0	imm9									0	0	Rn				Rt									
size										opc																									

STLURB <Wt>, [<Xn|SP>{, #<simm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

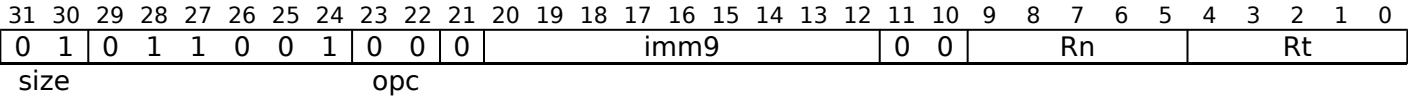
STLURH

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>{, #<simm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

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

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_ORDERED;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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](#). For information on single-copy atomicity and alignment requirements, see [Requirements for single-copy atomicity](#) and [Alignment of data accesses](#). If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics, as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

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

32-bit (sz == 0)

STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]

64-bit (sz == 1)

STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release
```

```
AccType acctype = if o0 == '1' then AccType_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        rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                X[t] = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
            else
                data = Mem[address, dbytes, acctype];
                X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

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

32-bit (size == 10)

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

64-bit (size == 11)

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC 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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
rn_unknown = FALSE;    // address is original base
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXR](#).

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is: 0 If the operation updates memory. 1 If the operation fails to update memory.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.


```

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

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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
            else
                data = Mem[address, dbytes, acctype];
                X[t] = ZeroExtend(data, regsize);

```


Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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 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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
rn_unknown = FALSE;    // address is original base

```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXRB](#).

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.


```

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

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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
                else
                    data = Mem[address, dbytes, acctype];
                    X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses see [Load/Store addressing modes](#).

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

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

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
rn_unknown = FALSE;    // address is original base

```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXRH](#).

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.


```

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

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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
                else
                    data = Mem[address, dbytes, acctype];
                    X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
1		x		1		1		1		0		0		0		0		0		0		imm9									0		1		Rn				Rt			
size										opc																																

32-bit (size == 10)

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

64-bit (size == 11)

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

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn				Rt									
size										opc																									

32-bit (size == 10)

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

64-bit (size == 11)

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

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
1		x		1		1		1		0		0		1		0		0		imm12											Rn				Rt			
size										opc																												

32-bit (size == 10)

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

64-bit (size == 11)

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

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

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<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);
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      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) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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);
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);
bits(64) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;
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;
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;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STRB \(immediate\)](#).

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <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);
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      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) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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>}]

```
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.
- <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, shift);
bits(64) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn				Rt					
size										opc																					

STRH <Wt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
0		1		1		1		1		0		0		1		0		0		imm12												Rn				Rt			
size										opc																													

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

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

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STRH \(immediate\)](#).

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <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, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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>}}]

```
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.
- <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);
bits(64) address;
bits(datasize) data;

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;

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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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*.

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

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

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

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

unpriv_at_el1 = PSTATE.EL == 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 = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>}]

```
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;

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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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);
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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	1	1	0	0	0	0	0	0	imm9									0	0	Rn				Rt									
size											opc																								

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

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
0	1	1	1	1	0	0	0	0	0	0	imm9									0	0	Rn						Rt								
size											opc																									

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

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = 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

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

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;

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

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

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

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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](#). For information on single-copy atomicity and alignment requirements, see [Requirements for single-copy atomicity](#) and [Alignment of data accesses](#). If a 64-bit pair 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](#).

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

32-bit (sz == 0)

STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]

64-bit (sz == 1)

STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
rn_unknown = FALSE;    // address is original base

```

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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
                else
                    data = Mem[address, dbytes, acctype];
                    X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). For information about memory accesses see [Load/Store addressing modes](#).

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

32-bit (size == 10)

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

64-bit (size == 11)

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC 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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
rn_unknown = FALSE;    // address is original base
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STXR](#).

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.


```

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

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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
            else
                data = Mem[address, dbytes, acctype];
                X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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						Rn						Rt			

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

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
rn_unknown = FALSE;    // address is original base

```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STXRB](#).

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.


```

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

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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                    X[t] = Mem[address + 0, 8, acctype];
                    X[t2] = Mem[address + 8, 8, acctype];
                else
                    data = Mem[address, dbytes, acctype];
                    X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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)										
size								L				o0				Rt2						Rn						Rt			

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

```

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
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 = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF       UNDEFINED;
            when Constraint_NOP         EndOfInstruction();
rn_unknown = FALSE;    // address is original base

```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
- 0** If the operation updates memory.

1

If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.


```

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

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];

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 = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
            X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t] = bits(datasize) UNKNOWN; // 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, AlignmentFault(acctype, iswrite, secondstage));
                X[t] = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
            else
                data = Mem[address, dbytes, acctype];
                X[t] = ZeroExtend(data, regsize);

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

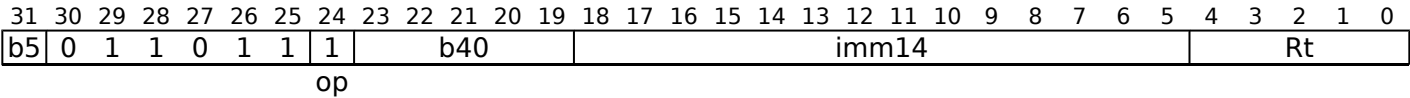
Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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.



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

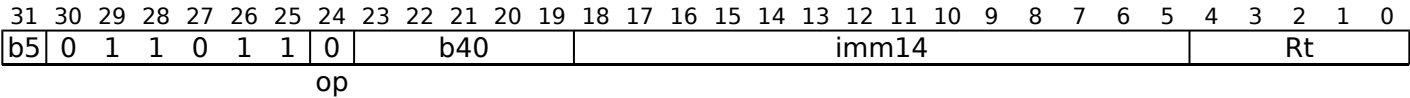
```
bits(datasize) operand = X[t];
boolean branch_conditional = TRUE;
if operand<bit_pos> == bit_val then
if operand<bit_pos> == bit_val then
boolean branch_conditional = TRUE;
BranchTo(PC[] + offset, BranchType_DIR, branch_conditional);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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.



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];
boolean branch_conditional = TRUE;
if operand<bit_pos> == bit_val then
if operand<bit_pos> == bit_val then
boolean branch_conditional = TRUE;
BranchTo(PC[] + offset, BranchType_DIR, branch_conditional);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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).

FMLSL, FMLSL2 (by element): Floating-point fused Multiply-Subtract Long from accumulator (by element).

FMLSL, FMLSL2 (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).

RSRA: 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 [v32.21](#)~~v32.15~~, AdvSIMD v29.05, pseudocode [v2021-06_xml](#)~~v2021-03~~, sve [v2021-06_rc2](#)~~v2021-03_rc2~~; Build timestamp: [2021-06-28T16:41:22](#)~~2021-03-30T21:41:22~~

Copyright © 2010-2021 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;

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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType fpcr    = FPCR[];
boolean merge     = fpop != 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 Elem[result, 0, esize] = FPSqrt(operand, fpcr);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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	f	t	y	p	e	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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTyp 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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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

```

CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(condition) then
    flags = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);
PSTATE.<N,Z,C,V> = flags;

```

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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(condition) then
    flags = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);
PSTATE.<N,Z,C,V> = flags;
```

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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

FCMP

Floating-point quiet Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the [PSTATE](#).{N, Z, C, V} flags.

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			0	0	1	0	0	0	Rn			0	x	0	0	0	
																												opc			

Half-precision (ftype == 11 && opc == 00) (FEAT_FP16)

FCMP <Hn>, <Hm>

Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 01) (FEAT_FP16)

FCMP <Hn>, #0.0

Single-precision (ftype == 00 && opc == 00)

FCMP <Sn>, <Sm>

Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 01)

FCMP <Sn>, #0.0

Double-precision (ftype == 01 && opc == 00)

FCMP <Dn>, <Dm>

Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 01)

FCMP <Dn>, #0.0

```
integer n = UInt(Rn);
integer m = UInt(Rm);    // ignored when opc<0> == '1'

integer datasize;
case 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 = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

Assembler Symbols

<Dn>	For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```

CheckFPEnabled64CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = if cmp_with_zero then FPZero('0') else V[m];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal all nans, FPCR[]);

```

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 $(\text{Operand1} < \text{Operand2})$, $(\text{Operand1} = \text{Operand2})$ and $(\text{Operand1} > \text{Operand2})$ are false. An unordered comparison sets the **PSTATE** condition flags to $N=0$, $Z=0$, $C=1$, and $V=1$.

Internal version only: isa ~~v32.21~~**v32.15**, AdvSIMD v29.05, pseudocode ~~v2021-06_xml~~**v2021-03**, sve ~~v2021-06_rc2b~~**v2021-03_rc2**; Build timestamp: ~~2021-06-28T16:22:11Z~~**2021-03-30T21:41:22Z**

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

FCMPE

Floating-point signaling Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the *PSTATE*.{N, Z, C, V} flags.

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	ftype	1	Rm				0	0	1	0	0	0	Rn				1	x	0	0	0			
opc																															

Half-precision (ftype == 11 && opc == 10) (FEAT_FP16)

FCMPE <Hn>, <Hm>

Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 11) (FEAT_FP16)

FCMPE <Hn>, #0.0

Single-precision (ftype == 00 && opc == 10)

FCMPE <Sn>, <Sm>

Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 11)

FCMPE <Sn>, #0.0

Double-precision (ftype == 01 && opc == 10)

FCMPE <Dn>, <Dm>

Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 11)

FCMPE <Dn>, #0.0

```
integer n = UInt(Rn);
integer m = UInt(Rm);    // ignored when opc<0> == '1'

integer datasize;
case 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 = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

Assembler Symbols

<Dn>	For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64CheckFPAdvSIMDEnabled64();  
bits(datasize) operand1 = V[n];  
bits(datasize) operand2;  
  
operand2 = if cmp_with_zero then FPZero('0') else V[m];  
  
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);
```

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 $(\text{Operand1} < \text{Operand2})$, $(\text{Operand1} = \text{Operand2})$ and $(\text{Operand1} > \text{Operand2})$ are false. An unordered comparison sets the **PSTATE** condition flags to $N=0$, $Z=0$, $C=1$, and $V=1$.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

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

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

Half-precision (ftype == 11) (FEAT_FP16)

FCSEL <Hd>, <Hn>, <Hm>, <cond>

Single-precision (ftype == 00)

FCSEL <Sd>, <Sn>, <Sm>, <cond>

Double-precision (ftype == 01)

FCSEL <Dd>, <Dn>, <Dm>, <cond>

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

bits(4) condition = cond;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
CheckFPEnabled64CheckFPAdvSIMDEnabled64();
bits(datasize) result;

result = if ConditionHolds(condition) then V[n] else V[m];

V[d] = result;
```

Operational information

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+01:004122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

FCVT

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the [FPCR](#) and writes the result to the SIMD&FP destination register.

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

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

Half-precision to single-precision (ftype == 11 && opc == 00)

FCVT <Sd>, <Hn>

Half-precision to double-precision (ftype == 11 && opc == 01)

FCVT <Dd>, <Hn>

Single-precision to half-precision (ftype == 00 && opc == 11)

FCVT <Hd>, <Sn>

Single-precision to double-precision (ftype == 00 && opc == 01)

FCVT <Dd>, <Sn>

Double-precision to half-precision (ftype == 01 && opc == 11)

FCVT <Hd>, <Dn>

Double-precision to single-precision (ftype == 01 && opc == 00)

FCVT <Sd>, <Dn>

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

if ftype == opc then UNDEFINED;

case ftype of
  when '00' srcsize = 32;
  when '01' srcsize = 64;
  when '10' UNDEFINED;
  when '11' srcsize = 16;
case opc of
  when '00' dstsize = 32;
  when '01' dstsize = 64;
  when '10' UNDEFINED;
  when '11' dstsize = 16;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();  
  
bits(srcsize) operand = V[n];  
FPCRTType fpcr      = FPCR[];  
boolean merge      = IsMerging(fpcr);  
bits(128) result = if merge then V[d] else Zeros();  
  
Elem[result, 0, dstsize] = FPConvert(operand, fpcr);  
  
V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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		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;
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

<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
```

```
FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;
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

<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
```

```
FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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		ftype		1		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</	

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'
    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

<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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.

31	30	29	28	27	26	25	24	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		1		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)

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'
    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

<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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.

31	30	29	28	27	26	25	24	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		0		0		0		Rn				Rd									
rmode																opcode																															

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'
    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

<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
```

```
FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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'
    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

<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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.

31	30	29	28	27	26	25	24	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		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	

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'
    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

<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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.

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

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'
    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

<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

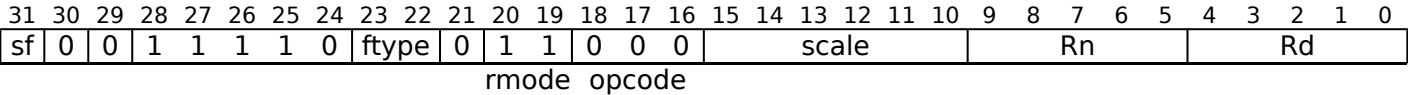
(new)

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;
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);

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;
  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

```

CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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, fracbits, 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, fracbits, unsigned, fpcr, rounding);
    V[d] = fltval;

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:21:41Z

Copyright © 2010-2021 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'
    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

<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;
FPCnvOp 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 = FPCnvOp_CVT_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR[]);
    unsigned = (opcode<0> == '1');
    op = FPCnvOp_CVT_ItoF;
  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

```

CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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, fracbits, 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, fracbits, unsigned, fpcr, rounding);
    V[d] = fltval;

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		0		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'
    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

<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

FDIV (scalar)

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

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

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

Half-precision (ftype == 11) (FEAT_FP16)

FDIV <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FDIV <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FDIV <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;
```

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

```
CheckFPEnabled64CheckFPAdvSIMDEnabled64();
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();

Elem[result, 0, esize] = FPDiv(operand1, operand2, FPCR[]);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

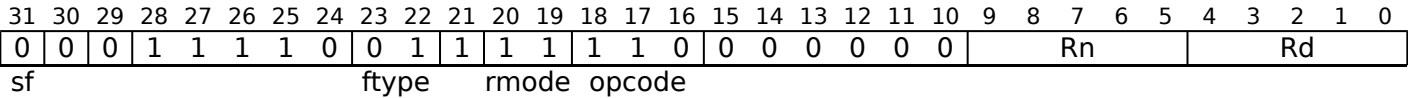
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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);

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

- <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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
```

```
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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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				
0001111								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;

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

```

CheckFPEnabled64CheckFPAdvSIMDEnabled64();

bits(esize) operandA = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType 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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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	ftype	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;

FPMaXMinOp operation;
case op of
    when '00' operation = FPMaXMinOp_MAX;
    when '01' operation = FPMaXMinOp_MIN;
    when '10' operation = FPMaXMinOp_MAXNUM;
    when '11' operation = FPMaXMinOp_MINNUM;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
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 FPMaxMinOp_MAX      Elem[result, 0, esize] = FPMax(operand1, operand2, fpcr);
  when FPMaxMinOp_MIN      Elem[result, 0, esize] = FPMin(operand1, operand2, fpcr);
  when FPMaxMinOp_MAXNUM   Elem[result, 0, esize] = FPMaxNum(operand1, operand2, fpcr);
  when FPMaxMinOp_MINNUM   Elem[result, 0, esize] = FPMinNum(operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;

FPMaXMinOp operation;
case op of
  when '00' operation = FPMaXMinOp_MAX;
  when '01' operation = FPMaXMinOp_MIN;
  when '10' operation = FPMaXMinOp_MAXNUM;
  when '11' operation = FPMaXMinOp_MINNUM;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
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();

case operation of
  when FPMAXMINOP_MAX    Elem[result, 0, esize] = FPMAX(operand1, operand2, fpcr);
  when FPMAXMINOP_MIN    Elem[result, 0, esize] = FPMIN(operand1, operand2, fpcr);
  when FPMAXMINOP_MAXNUM Elem[result, 0, esize] = FPMAXNUM(operand1, operand2, fpcr);
  when FPMAXMINOP_MINNUM Elem[result, 0, esize] = FPMINNUM(operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21-03-30T21:4122

Copyright © 2010-2021 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	f	t	y	p	e	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;

FPMaXMinOp operation;
case op of
    when '00' operation = FPMaXMinOp_MAX;
    when '01' operation = FPMaXMinOp_MIN;
    when '10' operation = FPMaXMinOp_MAXNUM;
    when '11' operation = FPMaXMinOp_MINNUM;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
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 FPMaxMinOp_MAX      Elem[result, 0, esize] = FPMax(operand1, operand2, fpcr);
  when FPMaxMinOp_MIN      Elem[result, 0, esize] = FPMin(operand1, operand2, fpcr);
  when FPMaxMinOp_MAXNUM   Elem[result, 0, esize] = FPMaxNum(operand1, operand2, fpcr);
  when FPMaxMinOp_MINNUM   Elem[result, 0, esize] = FPMinNum(operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;

FPMaXMinOp operation;
case op of
  when '00' operation = FPMaXMinOp_MAX;
  when '01' operation = FPMaXMinOp_MIN;
  when '10' operation = FPMaXMinOp_MAXNUM;
  when '11' operation = FPMaXMinOp_MINNUM;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
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();

case operation of
  when FPMaxMinOp_MAX      Elem[result, 0, esize] = FPMax(operand1, operand2, fpcr);
  when FPMaxMinOp_MIN      Elem[result, 0, esize] = FPMin(operand1, operand2, fpcr);
  when FPMaxMinOp_MAXNUM    Elem[result, 0, esize] = FPMaxNum(operand1, operand2, fpcr);
  when FPMaxMinOp_MINNUM    Elem[result, 0, esize] = FPMinNum(operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+00:00 4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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	f	t	y	p	e	1	0	0	0	0	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;

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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType fpcr    = FPCR[];
boolean merge     = fpop != 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 Elem[result, 0, esize] = FPSqrt(operand, fpcr);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

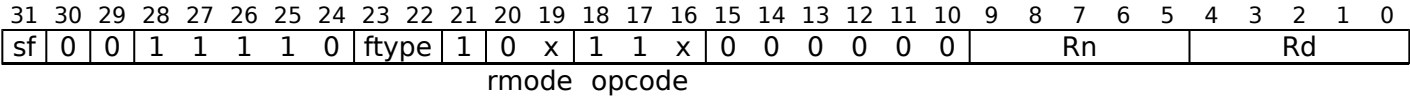
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(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

```

CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

FMOV (scalar, immediate)

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD&FP destination register.

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	f	t	y	p	e	1	imm8						1	0	0	0	0	0	0	Rd				

Half-precision (ftype == 11) (FEAT_FP16)

FMOV <Hd>, #<imm>

Single-precision (ftype == 00)

FMOV <Sd>, #<imm>

Double-precision (ftype == 01)

FMOV <Dd>, #<imm>

```
integer d = UInt(Rd);
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;
bits(datasize) imm = VFPEExpandImm(imm8);
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in A64 floating-point instructions](#).

Operation

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
V[d] = imm;
```

(old)

htmldiff from-

(new)

FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

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

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

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

```

CheckFPEnabled64CheckFPAdvSIMDEnabled64();

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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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.

31	30	29	28	27	26	25	24	23	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	0	0	0	1	0	Rn			Rd					
op																															

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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;

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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTyp fpcr    = FPCR[];
boolean merge    = fpop != 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 Elem[result, 0, esize] = FPSqrt(operand, fpcr);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;

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

```

CheckFPEnabled64CheckFPAdvSIMDEnabled64();

bits(esize) operandA = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTyp 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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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;

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

```

CheckFPEnabled64CheckFPAdvSIMDEnabled64();

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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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	t	y	p	e	1		R	m	1	0	0	0	1	0		R	n			R	d		
																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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

FRECPX

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

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

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision (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	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn					Rd				

FRECPX <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;
```

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

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

```
integer esize = 16;integer esize = 16;
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

Single-precision and double-precision

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

FRECPX <V><d>, <V><n>

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

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

```
integer esize = 32 << UInt(sz);(sz);
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) operand =bits(datasize) operand = V[n];

FPCRType fpcr    = FPCR[];
boolean merge    =boolean merge    = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();();
bits(esize) element;

for e = 0 to elements-1
    element =

Elem[result, 0, esize] =operand, e, esize]; Elem[result, e, esize] = FPrecpX(operand, fpcr);(element, fp
V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

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, intsize);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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, intsize);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

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, intsize);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;

integer intsize = if op<1> == '0' then 32 else 64;

FPRounding rounding = if op<0> == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

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, intsize);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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	1	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;

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

```
CheckFPEnabled64CheckFPAdvSIMDEnabled64();

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, exact);

V[d] = result;
```

Internal version only: isa v32.21v32-15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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	ftype			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;
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

```
CheckFPEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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, exact);

V[d] = result;
```

Internal version only: isa v32.21v32-15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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	ftype	1	0	0	1	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;
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

```
CheckFPEnabled64CheckFPAdvSIMDEnabled64();

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, exact);

V[d] = result;
```

Internal version only: isa v32.21v32-15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;
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

```
CheckFPEnabled64CheckFPAdvSIMDEnabled64();

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, exact);

V[d] = result;
```

Internal version only: isa v32.21v32-15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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	ftype				1	0	0	1	0	0	1	1	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;
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

```
CheckFPEnabled64CheckFPAdvSIMDEnabled64();

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, exact);

V[d] = result;
```

Internal version only: isa v32.21v32-15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;

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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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, exact);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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;

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

```
CheckFPEnabled64CheckFPAdvSIMDEnabled64();  
  
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, exact);  
  
V[d] = result;
```

Internal version only: isa v32.21v32-15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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	f	t	y	p	e	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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTyp fpcr    = FPCR[];
boolean merge   = fpop != 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 Elem[result, 0, esize] = FPSqrt(operand, fpcr);

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

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

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

Half-precision (ftype == 11) (FEAT_FP16)

FSUB <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FSUB <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FSUB <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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTyp 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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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)

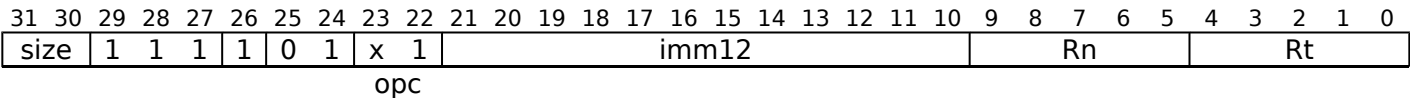
```
LDR <Dt>, [<Xn|SP>, #<sim>]!
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<l>: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<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

```
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64(tag_checked);
();
bits(64) address;
bits(datasize) data;

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

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 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);
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

```
CheckFPAdvSIMDEnabled64();

bits(64) address = PC[] + offset;
bits(size*8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64(tag_checked);
();

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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64(tag_checked);
();
bits(64) address;
bits(datasize) data;

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

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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<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 && (wback || n != 31);
```

Operation

```
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64(tag_checked);
();
bits(64) address;
bits(datasize) data;

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

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

PMULL, PMULL2

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}*.

The PMULL instruction extracts each source vector from the lower half of each source register. The PMULL2 instruction extracts each source vector from the upper half of each source register.

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

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

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

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

if size == '01' || size == '10' then UNDEFINED;
if size == '11' && !HaveBit128PMULLExt() then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

size	<Ta>
00	8H
01	RESERVED
10	RESERVED
11	1Q

The '1Q' arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

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

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

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, 2*esize] = PolynomialMult(element1, element2);

V[d] = result;
```

Operational information

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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>;
    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 v32.21v32.45, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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 extracts writes each the source vector from to the lower half of each the source destination register and clears the upper half, while the SABDL2 instruction extracts writes each the source vector from to the upper half of each the source 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>;
    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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

SADDL, SADDL2

Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SADDL instruction extracts each source vector from the lower half of each source register, while the SADDL2 instruction extracts each source vector from the upper half of each source register.

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

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

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

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

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

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

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

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

SADDW, SADDW2

Signed Add Wide. This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The SADDW instruction extracts the second source vector from the lower half of the second source register, while the SADDW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

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

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

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

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

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

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

Operation

```

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

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

V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

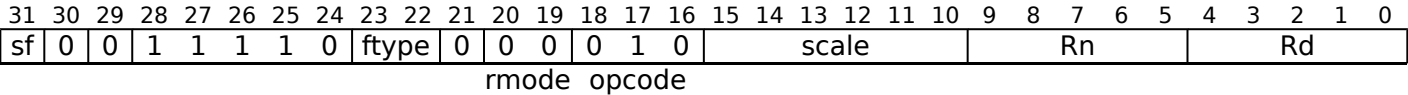
(new)

SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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



**32-bit to half-precision (sf == 0 && 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);

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;
  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

```

CheckFPEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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, fracbits, 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, fracbits, unsigned, fpcr, rounding);
    V[d] = fltval;

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

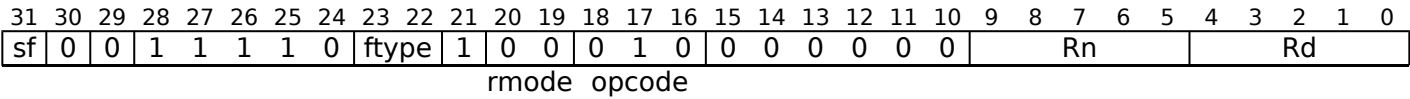
(new)

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'
    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.
<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();
```

```
FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

SHLL, SHLL2

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SHLL instruction extracts vector elements from the lower half of the source register, while the SHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

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

SHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

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

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

integer shift = esize;
boolean unsigned = FALSE; // Or TRUE without change of functionality
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

<shift> Is the left shift amount, which must be equal to the source element width in bits, encoded in “size”:

size	<shift>
00	8
01	16
10	32
11	RESERVED

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+01:00 4122

Copyright © 2010-2021 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];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

    for i = 0 to 3
        Elem[intval,i,8] = Sbox(Elem[intval,i,8]);

    intval = intval EOR ROL(intval,2) EOR ROL(intval,10) EOR ROL(intval,18) EOR ROL(intval,24);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;

V[d] = roundresult;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

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

Internal version only: isa ~~v32.21~~v32.15, AdvSIMD v29.05, pseudocode ~~v2021-06_xml~~v2021-03, sve ~~v2021-06_rc2~~v2021-03_rc2 ; Build timestamp: ~~2021-06-28T16:20:21-03-30T21:41:22~~

Copyright © 2010-2021 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(32) intval;
bits(8) sbxout;
bits(128) result;
bits(32) const;
bits(128) roundresult;

roundresult = V[n];
for index = 0 to 3
    const = Elem[Vm,index,32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;

    for i = 0 to 3
        Elem[intval,i,8] = Sbox(Elem[intval,i,8]);

    intval = intval EOR ROL(intval,13) EOR ROL(intval,23);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;

V[d] = roundresult;
```


Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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				Rn					Rd	
U										o2																					

SMLAL{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');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

(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. 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>;
    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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+00:00 2021-03-30T21:41:22

Copyright © 2010-2021 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 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');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

(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>;
    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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+00:00 2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts vector elements from the lower half of the first source register, while the SMULL2 instruction extracts vector elements from the upper half of the first source register.

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

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

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 <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) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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. 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>;

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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 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.~~register, ~~The~~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	
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) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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](#). [The](#) 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(0);
integer elements = datasize DIV esize;

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

- <Ta> Is an arrangement specifier, encoded in "size":

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

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

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

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

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

- <Va> Is the destination width specifier, encoded in "size":

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

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

- <Vb> Is the source width specifier, encoded in "size":

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

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

- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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.register*, *Thewhile 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					Rn				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	
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(idxsized)  operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v32.21-v32.15, AdvSIMD v29.05, pseudocode v2021-06_xml-v2021-03, sve v2021-06_rc2bv2021-03_rc2; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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](#). [The](#) 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(0);
integer elements = datasize DIV esize;

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

- <Ta> Is an arrangement specifier, encoded in "size":

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

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

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

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

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

- <Va> Is the destination width specifier, encoded in "size":

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

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

- <Vb> Is the source width specifier, encoded in "size":

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

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

- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2*esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2*esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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. The 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) = SignedSatQ(2 * element1 * element2, 2*esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21-03-30T21:4122

Copyright © 2010-2021 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);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;
```

(old)

htmldiff from-

(new)

SSHLL, SSHLL2

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SSHLL instruction extracts vector elements from the lower half of the source register, while the SSHLL2 instruction extracts vector elements from the upper half of the source register.

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

This instruction is used by the alias [SXTL](#), [SXTL2](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	1	0	0	1	Rn				Rd					
U									immh																						

SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

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

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

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

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

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	RESERVED

Alias Conditions

Alias	Is preferred when
SXTL, SXTL2	immb == '000' && BitCount (immh) == 1

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa [v32.21](#)~~v32.15~~, AdvSIMD v29.05, pseudocode [v2021-06_xml](#)~~v2021-03~~, sve [v2021-06_rc2](#)~~v2021-03_rc2~~; Build timestamp: [2021-06-28T16:41:22](#)~~2021-03-30T21:41:22~~

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

SSUBL, SSUBL2

Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The SSUBL instruction extracts each source vector from the lower half of each source register, while the SSUBL2 instruction extracts each source vector from the upper half of each source register.

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

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

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

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

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

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

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

V[d] = result;
```

Operational information

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

SSUBW, SSUBW2

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The SSUBW instruction extracts the second source vector from the lower half of the second source register, while the SSUBW2 instruction extracts the second source vector from the upper half of the second source register.

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

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

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

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

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

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

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

Operation

```

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

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

V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21-03-30T21:41:22

Copyright © 2010-2021 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)

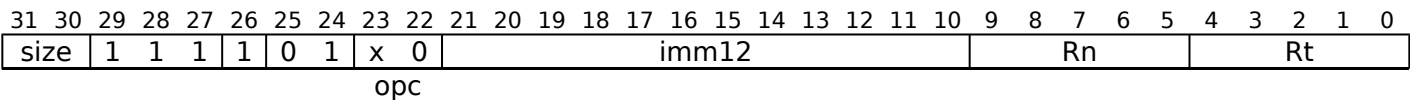
```
STR <Dt>, [<Xn|SP>, #<sim>]!
```

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

```
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64(tag_checked);
();
bits(64) address;
bits(datasize) data;

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

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 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	SXTX

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	SXTX

<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
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64(tag_checked);
();
bits(64) address;
bits(datasize) data;

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

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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 && (wback || n != 31);
```

Operation

```
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64(tag_checked);
();
bits(64) address;
bits(datasize) data;

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

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

SXTL, SXTL2

Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values. The SXTL instruction extracts the source vector from the lower half of the source register, while the SXTL2 instruction extracts the source vector from the upper half of the source register. Depending on the settings in the [CPACR EL1](#), [CPTR EL2](#), and [CPTR EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of [SSHLL, SSHLL2](#). This means:

- The encodings in this description are named to match the encodings of [SSHLL, SSHLL2](#).
- The description of [SSHLL, SSHLL2](#) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				0	0	0	1	0	1	0	0	1	Rn					Rd				
U									immh				immb																		

SXTL{2} <Vd>.<Ta>, <Vn>.<Tb>

is equivalent to

SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #0

and is the preferred disassembly when `BitCount(immh) == 1`.

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

<Ta> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

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

<Tb> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

Operation

The description of [SSHLL](#), [SSHLL2](#) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa [v32.21](#)~~v32.15~~, AdvSIMD v29.05, pseudocode [v2021-06_xml](#)~~v2021-03~~, sve [v2021-06_rc2](#)~~v2021-03_rc2~~ ; Build timestamp: [2021-06-28T16:41:22](#)~~2021-03-30T21:41:22~~

Copyright © 2010-2021 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. The 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>;
    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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 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>;
    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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

UADDL, UADDL2

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UADDL instruction extracts each source vector from the lower half of each source register, while the UADDL2 instruction extracts each source vector from the upper half of each source register.

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

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

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

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

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

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

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

V[d] = result;
```

Operational information

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

UADDW, UADDW2

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The UADDW instruction extracts vector elements from the lower half of the second source register, while the UADDW2 instruction extracts vector elements from the upper half of the second source register.

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

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

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

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

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

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

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

Operation

```

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

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

V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+01:00

Copyright © 2010-2021 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);

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;
  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

```

CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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, fracbits, 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, fracbits, unsigned, fpcr, rounding);
    V[d] = fltval;

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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.

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

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'
    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.
<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

```
CheckFPEEnabled64CheckFPAdvSIMDEnabled64();

FPCRTType 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];
    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, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 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 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');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

(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>;
    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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+00:00 2021-03-30T21:41:22

Copyright © 2010-2021 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 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');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(idxsizesize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

(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. 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>;
    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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+00:00 2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMULL instruction extracts vector elements from the lower half of the first source register, while the UMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the [CPACR_EL1](#), [CPTR_EL2](#), and [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

<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) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

UMULL, UMULL2 (vector)

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMULL instruction extracts each source vector from the lower half of each source register, while the UMULL2 instruction extracts each source vector from the upper half of each source register.

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

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

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>;

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 v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

USHLL, USHLL2

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The USHLL instruction extracts vector elements from the lower half of the source register, while the USHLL2 instruction extracts vector elements from the upper half of the source register.

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

This instruction is used by the alias [UXTL](#), [UXTL2](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	1	0	0	1	Rn				Rd					
U									immh																						

USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

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

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

<Ta> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

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

<Tb> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	RESERVED

Alias Conditions

Alias	Is preferred when
UXTL, UXTL2	immb == '000' && BitCount (immh) == 1

Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa [v32.21](#)~~v32.15~~, AdvSIMD v29.05, pseudocode [v2021-06_xml](#)~~v2021-03~~, sve [v2021-06_rc2](#)~~v2021-03_rc2~~; Build timestamp: [2021-06-28T16:41:22](#)~~2021-03-30T21:41:22~~

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

USUBL, USUBL2

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The USUBL instruction extracts each source vector from the lower half of each source register, while the USUBL2 instruction extracts each source vector from the upper half of each source register.

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

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

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

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

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

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)  operand1 = Vpart[n, part];
bits(datasize)  operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

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

V[d] = result;
```

Operational information

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

USUBW, USUBW2

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are ~~signed~~ **unsigned** integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The USUBW instruction extracts vector elements from the lower half of the first source ~~register~~ **register**, ~~while the~~ **while the** USUBW2 instruction extracts vector elements from the upper half of the first source register.

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

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

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

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

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

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

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

Operation

```

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

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

V[d] = result;

```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+01:00

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

UXTL, UXTL2

Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The UXTL instruction extracts vector elements from the lower half of the source register, while the UXTL2 instruction extracts vector elements from the upper half of the source register.

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

This is an alias of [USHLL](#), [USHLL2](#). This means:

- The encodings in this description are named to match the encodings of [USHLL](#), [USHLL2](#).
- The description of [USHLL](#), [USHLL2](#) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	1	0	1	1	1	1	0	!= 0000				0	0	0	1	0	1	0	0	1	Rn						Rd					
U									immh				immb																				

UXTL{2} <Vd>.<Ta>, <Vn>.<Tb>

is equivalent to

USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #0

and is the preferred disassembly when `BitCount(immh) == 1`.

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

<Ta> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

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

<Tb> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

Operation

The description of [USHLL](#), [USHLL2](#) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa **v32.21**~~v32.15~~, AdvSIMD v29.05, pseudocode **v2021-06_xml**~~v2021-03~~, sve **v2021-06_rc2b**~~v2021-03_rc2~~ ; Build timestamp: **2021-06-28T16:41:22**~~2021-03-30T21:41:22~~

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

(old)

htmldiff from-

(new)

A64 -- SVE Instructions (alphabetic order)

ABS: Absolute value (predicated).

ADD (immediate): Add immediate (unpredicated).

ADD (vectors, predicated): Add vectors (predicated).

ADD (vectors, unpredicated): Add vectors (unpredicated).

ADDPL: Add multiple of predicate register size to scalar register.

ADDVL: Add multiple of vector register size to scalar register.

ADR: Compute vector address.

AND (immediate): Bitwise AND with immediate (unpredicated).

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

AND (vectors, predicated): Bitwise AND vectors (predicated).

AND (vectors, unpredicated): Bitwise AND vectors (unpredicated).

[ANDS](#)~~AND, ANDS (predicates)~~: Bitwise AND [predicates, setting the condition flags](#).~~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).

[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.

BIC (immediate): Bitwise clear bits using immediate (unpredicated): an alias of AND (immediate).

[BIC \(predicates\)](#): Bitwise clear predicates.

BIC (vectors, predicated): Bitwise clear vectors (predicated).

BIC (vectors, unpredicated): Bitwise clear vectors (unpredicated).

~~BICSBIC, BICS (predicates)~~: Bitwise clear ~~predicates, setting the condition flags.~~~~predicates.~~

~~BRKA, BRKAS~~: Break after first true condition.

~~BRKAS~~: Break after first true condition, setting the condition flags.

~~BRKB, BRKBS~~: Break before first true condition.

~~BRKBS~~: Break before first true condition, setting the condition flags.

~~BRKN, BRKNS~~: Propagate break to next partition.

~~BRKNS~~: Propagate break to next partition, setting the condition flags.

~~BRKPA, BRKPAS~~: Break after first true condition, propagating from previous partition.

~~BRKPAS~~: Break after first true condition, propagating from previous partition and setting the condition flags.

~~BRKPB, BRKPBS~~: Break before first true condition, propagating from previous partition.

~~BRKPBS~~: Break before first true condition, propagating from previous partition and setting the condition flags.

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).

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 (predicates): Bitwise exclusive OR predicates.

EOR (vectors, predicated): Bitwise exclusive OR vectors (predicated).

EOR (vectors, unpredicated): Bitwise exclusive OR vectors (unpredicated).

~~EORSEOR, EORS (predicates)~~: Bitwise exclusive OR **predicates, setting the condition flags.** ~~predicates.~~

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.

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).

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.

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).

FMAXNMV: Floating-point maximum number recursive reduction to scalar.

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).

FMINNMV: Floating-point minimum number recursive reduction to scalar.

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 [$Z_{da} = Z_{da} + Z_n * Z_m$].

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 [$Z_{da} = Z_{da} + -Z_n * Z_m$].

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).

FNMAD: 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 [$Z_{da} = -Z_{da} + Z_n * Z_m$].
 FNMSB: Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand [$Z_{dn} = -Z_a + Z_{dn} * Z_m$].
 FRECPPE: Floating-point reciprocal estimate (unpredicated).
 FRECPSP: Floating-point reciprocal step (unpredicated).
 FRECPXP: Floating-point reciprocal exponent (predicated).
 FRINT<r>: Floating-point round to integral value (predicated).
 FRSQRTE: Floating-point reciprocal square root estimate (unpredicated).
 FRSQRTPS: 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.
 FTSSSEL: Floating-point trigonometric select coefficient.
 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).

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).

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).

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).

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 [$Z_{dn} = Z_a + Z_{dn} * Z_m$].

MLA: Multiply-add vectors (predicated), writing addend [$Z_{da} = Z_{da} + Z_n * Z_m$].

MLS: 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 ANDS, AND, ANDS (predicates).

MOVS (unpredicated): Move predicate (unpredicated), setting the condition flags: an alias of ORRS, ORR, ORRS (predicates).

MSB: Multiply-subtract vectors (predicated), writing multiplicand [$Zdn = Za - Zdn * Zm$].

MUL (immediate): Multiply by immediate (unpredicated).

MUL (vectors): Multiply vectors (predicated).

NAND, NANDS: Bitwise NAND predicates.

NANDS: Bitwise NAND predicates, setting the condition flags.

NEG: Negate (predicated).

NOR, NORS: Bitwise NOR predicates.

NORS: Bitwise NOR predicates, setting the condition flags.

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 EORS, 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.

ORNS: Bitwise inclusive OR inverted predicate, setting the condition flags.

ORR (immediate): Bitwise inclusive OR with immediate (unpredicated).

ORR (predicates): Bitwise inclusive OR predicates.

ORR (vectors, predicated): Bitwise inclusive OR vectors (predicated).

ORR (vectors, unpredicated): Bitwise inclusive OR vectors (unpredicated).

ORRS, ORR, ORRS (predicates): Bitwise inclusive OR predicates, setting the condition flags, predicate.

ORV: Bitwise inclusive OR reduction to scalar.

PFALSE: Set all predicate elements to false.

PFIRST: Set the first active predicate element to true.

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.

PTRUES: Initialise predicate from named constraint and set the condition flags.

PUNPKHI, PUNPKLO: Unpack and widen half of predicate.

RBIT: Reverse bits (predicated).

~~RDFFR (unpredicated): Read the first-fault register.~~

~~RDFFR, RDFSRS~~ (predicated): Return predicate of successfully loaded elements.

RDFFR (unpredicated): Read the first-fault register.

RDFSRS: Return predicate of successfully loaded elements, setting the condition flags.

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, REWV: Reverse bytes / halfwords / words within elements (predicated).

SABD: Signed absolute difference (predicated).

SADDV: Signed add reduction to scalar.

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.

SMAX (immediate): Signed maximum with immediate (unpredicated).

SMAX (vectors): Signed maximum vectors (predicated).

SMAV: Signed maximum reduction to scalar.

SMIN (immediate): Signed minimum with immediate (unpredicated).

SMIN (vectors): Signed minimum vectors (predicated).

SMINV: Signed minimum reduction to scalar.

SMMLA: Signed integer matrix multiply-accumulate.

SMULH: Signed multiply returning high half (predicated).

SPLICE: Splice two vectors under predicate control.

SQADD (immediate): Signed saturating add immediate (unpredicated).

SQADD (vectors): Signed saturating add vectors (unpredicated).

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.

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.

SQSUB (immediate): Signed saturating subtract immediate (unpredicated).

SQSUB (vectors): Signed saturating subtract vectors (unpredicated).

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).

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).

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).

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).

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).

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.

SXTB, SXTH, SXTW: Signed byte / halfword / word extend (predicated).

TBL: Programmable table lookup in single vector table.

TRN1, TRN2 (predicates): Interleave even or odd elements from two predicates.

[TRN1, TRN2 \(vectors\)](#): Interleave even or odd elements from two vectors.

UABD: Unsigned absolute difference (predicated).

UADDV: Unsigned add reduction to scalar.

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.

UMAX (immediate): Unsigned maximum with immediate (unpredicated).

UMAX (vectors): Unsigned maximum vectors (predicated).

UMAXV: Unsigned maximum reduction to scalar.

UMIN (immediate): Unsigned minimum with immediate (unpredicated).

UMIN (vectors): Unsigned minimum vectors (predicated).

UMINV: Unsigned minimum reduction to scalar.

[UMMLA](#): Unsigned integer matrix multiply-accumulate.

UMULH: Unsigned multiply returning high half (predicated).

UQADD (immediate): Unsigned saturating add immediate (unpredicated).

UQADD (vectors): 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.

UQSUB (immediate): Unsigned saturating subtract immediate (unpredicated).

UQSUB (vectors): Unsigned saturating subtract vectors (unpredicated).

[USDOT \(indexed\)](#): Unsigned by signed integer indexed dot product.

[USDOT \(vectors\)](#): Unsigned by signed integer dot product.

[USMMLA](#): Unsigned by signed integer matrix multiply-accumulate.

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.

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.

WRFFR: Write the first-fault register.

ZIP1, ZIP2 (predicates): Interleave elements from two half predicates.

[ZIP1, ZIP2 \(vectors\)](#): Interleave elements from two half vectors.

Internal version only: isa [v32.21](#)~~v32.15~~, AdvSIMD v29.05, pseudocode [v2021-06_xml](#)~~v2021-03~~, sve [v2021-06_rc2](#)~~v2021-03_rc2~~; Build timestamp: [2021-06-28T16:41:22](#)~~2021-03-30T21:41:22~~

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

AND, ANDS (predicates)

Bitwise AND predicates

Not setting the condition flags

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the [alias](#)[aliases](#) [MOVS \(predicated\)](#), and [MOV \(predicate, predicated, zeroing\)](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	0	0	0	Pm			0	1	Pg			0	Pn			0	Pd						
S																															

AND <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	0	1	0	0		Pm			0	1		Pg		0		Pn		0			Pd			
S																															

ANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
MOV (predicate, predicated, zeroing)	MOVS (predicated)
	S == '01' && Pn == Pm

Alias

MOV (predicate, predicated, zeroing)

Is preferred when

S == '0' && Pn == Pm

Operation

```

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

ANDS

Bitwise AND predicates, setting the condition flags

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias [MOVS \(predicated\)](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	1	0	0		Pm		0	1		Pg		0		Pn		0		Pd					
S																															

ANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
MOVS (predicated)	S == '1' && Pn == Pm

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

no old file

htmldiff from-

(new)

BFCVT

Floating-point down convert to BFloat16 format (predicated)

Convert to BFloat16 from single-precision in each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Since the result type is smaller than the input type, the results are zero-extended to fill each destination element. Unlike the BFloat16 matrix multiplication and dot product instructions, this instruction honors all of the FPCR bits that apply to single-precision arithmetic. 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_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(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	1	1	0	0	1	0	1	1	0	0	0	1	0	1	0	1	0	1	Pg			Zn			Zd						

BFCVT <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 32;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, 32) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
    if ElemP[mask, e, 32] == '1' then
        bits(32) element = Elem[operand, e, 32];
        Elem[result, 2*e, 16] = FPConvertBF(element, FPCR[]);
        Elem[result, 2*e+1, 16] = Zeros();

Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa ~~v32.21~~v32.15, AdvSIMD v29.05, pseudocode ~~v2021-06_xml~~v2021-03, sve ~~v2021-06_rc2~~v2021-03-rc2 ; Build timestamp: ~~2021-06-28T16:20:21-03-30T21:41:22~~

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

BFCVTNT

Floating-point down convert and narrow to BFloat16 (top, predicated)

Convert active 32-bit single-precision elements from the source vector to BFloat16 format, and place the results in the odd-numbered 16-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.

Unlike the BFloat16 matrix multiplication and dot product instructions, this instruction honors all of the FPCR bits that apply to single-precision arithmetic. 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_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(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	1	1	0	0	1	0	0	1	0	0	0	1	0	1	0	1	0	1	Pg			Zn			Zd						

BFCVTNT <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 32;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, 32) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
    if ElemP[mask, e, 32] == '1' then
        bits(32) element = Elem[operand, e, 32];
        Elem[result, 2*e+1, 16] = FPConvertBF(element, FPCR[]);

Z[d] = result;
```

BFDOT (vectors)

BFloat16 floating-point dot product

The BFloat16 floating-point (BF16) dot product instruction computes the dot product of a pair of BF16 values held in each 32-bit element of the first source vector multiplied by a pair of BF16 values in the corresponding 32-bit element of the second source vector, and then destructively adds the single-precision dot product to the corresponding single-precision element of the destination vector.

This instruction is unpredicated.

All floating-point calculations performed by this instruction are performed with the following behaviors, irrespective of the value in FPCR:

- * Uses the non-IEEE 754 Round-to-Odd mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- * The cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC and IOC) are not modified.
- * Trapped floating-point exceptions are disabled, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE and IOE) are all zero.
- * Denormalized inputs and results are flushed to zero, as if FPCR.FZ == 1.
- * Only the Default NaN is generated, as if FPCR.DN == 1.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(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	1	1	0	0	1	0	0	0	1	1	Zm				1	0	0	0	0	0	Zn				Zda						

BFDOT <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];

    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da] = result;
```


Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

BFDOT (indexed)

BFloat16 floating-point indexed dot product

The BFloat16 floating-point (BF16) indexed dot product instruction computes the dot product of a pair of BF16 values held in each 32-bit element of the first source vector multiplied by a pair of BF16 values in an indexed 32-bit element of the second source vector, and then destructively adds the single-precision dot product to the corresponding single-precision element of the destination vector.

The BF16 pairs within the second source vector are specified using an immediate index which selects the same BF16 pair position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

All floating-point calculations performed by this instruction are performed with the following behaviors, irrespective of the value in FPCR:

- * Uses the non-IEEE 754 Round-to-Odd mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- * The cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC and IOC) are not modified.
- * Trapped floating-point exceptions are disabled, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE and IOE) are all zero.
- * Denormalized inputs and results are flushed to zero, as if FPCR.FZ == 1.
- * Only the Default NaN is generated, as if FPCR.DN == 1.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(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	1	1	0	0	1	0	0	0	1	1	i2		Zm		0	1	0	0	0	0			Zn					Zda			

BFDOT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i2);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 32;
integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];

    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+00:004122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

BFMLALB (vectors)

BFloat16 floating-point multiply-add long to single-precision (bottom)

This BFloat16 floating-point multiply-add long instruction widens the even-numbered 16-bit BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

Unlike the BFloat16 matrix multiplication and dot product instructions, this instruction performs a fused multiply-add that honors all of the FPCR bits that apply to single-precision arithmetic. 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_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(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	1	1	0	0	1	0	0	1	1	1	Zm				1	0	0	0	0	0	0	Zn				Zda					
o2										op					T																

BFMLALB <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + 0, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, 2 * e + 0, 16] : Zeros(16);
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(element3, element1, element2, FPCR[]);

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:21+00:004122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

BFMLALB (indexed)

BFloat16 floating-point multiply-add long to single-precision (bottom, indexed)

This BFloat16 floating-point multiply-add long instruction widens the even-numbered 16-bit BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

Unlike the BFloat16 matrix multiplication and dot product instructions, this instruction performs a fused multiply-add that honors all of the FPCR bits that apply to single-precision arithmetic. 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_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(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	1	1	0	0	1	0	0	1	1	1	i3h		Zm		0	1	0	0	i3l	0				Zn					Zda		
o2										op										T											

BFMLALB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 32;
integer eltsegment = 128 DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltsegment);
    integer s = 2 * segmentbase + index;
    bits(32) element1 = Elem[operand1, 2 * e + 0, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, s, 16] : Zeros(16);
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMLAdd(element3, element1, element2, FPCR[]);

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2hv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

BFMLALT (vectors)

BFloat16 floating-point multiply-add long to single-precision (top)

This BFloat16 floating-point multiply-add long instruction widens the odd-numbered 16-bit BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

Unlike the BFloat16 matrix multiplication and dot product instructions, this instruction performs a fused multiply-add that honors all of the FPCR bits that apply to single-precision arithmetic. 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_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(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	1	1	0	0	1	0	0	1	1	1	Zm				1	0	0	0	0	1	Zn				Zda						
o2										op					T																

BFMLALT <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + 1, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, 2 * e + 1, 16] : Zeros(16);
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(element3, element1, element2, FPCR[]);

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:21+00:004122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

BFMLALT (indexed)

BFloat16 floating-point multiply-add long to single-precision (top, indexed)

This BFloat16 floating-point multiply-add long instruction widens the odd-numbered 16-bit BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated. Unlike the BFloat16 matrix multiplication and dot product instructions, this instruction performs a fused multiply-add that honors all of the FPCR bits that apply to single-precision arithmetic. 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_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(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	1	1	0	0	1	0	0	1	1	1	i3h		Zm		0	1	0	0	i3l	1				Zn					Zda		
o2										op										T											

BFMLALT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 32;
integer eltsperssegment = 128 DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltsperssegment);
    integer s = 2 * segmentbase + index;
    bits(32) element1 = Elem[operand1, 2 * e + 1, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, s, 16] : Zeros(16);
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(element3, element1, element2, FPCR[]);

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2hv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

BFMMLA

BFloat16 floating-point matrix multiply-accumulate

This BFloat16 floating-point (BF16) matrix multiply-accumulate instruction multiplies the 2×4 matrix of BF16 values held in each 128-bit segment of the first source vector by the 4×2 BF16 matrix in the corresponding segment of the second source vector. The resulting 2×2 single-precision (FP32) matrix product is then destructively added to the FP32 matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing a 4-way dot product per destination element.

This instruction is unpredicated and vector length agnostic.

All floating-point calculations performed by this instruction are performed with the following behaviors, irrespective of the value in FPCR:

- * Uses the non-IEEE 754 Round-to-Odd mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- * The cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC and IOC) are not modified.
- * Trapped floating-point exceptions are disabled, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE and IOE) are all zero.
- * Denormalized inputs and results are flushed to zero, as if FPCR.FZ == 1.
- * Only the Default NaN is generated, as if FPCR.DN == 1.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(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	1	1	0	0	1	0	0	0	1	1	Zm				1	1	1	0	0	1	Zn				Zda						

BFMMLA <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res = BFMatMulAdd(addend, op1, op2);
    Elem[result, s, 128] = res;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2hv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

BIC, BICS (predicates)

Bitwise clear predicates

Not setting the condition flags

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	0	0	0	Pm			0	1	Pg			0	Pn			1	Pd						
S																															

BIC <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	0	1	0	0		Pm			0	1		Pg			0			Pn		1			Pd	

S

BICS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+01:004122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

BICS

Bitwise clear predicates, setting the condition flags

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	1	0	0	Pm			0		1	Pg			0		Pn			1		Pd			
S																															

BICS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BRKA, BRKAS

Break after first true condition

Not setting the condition flags

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Does not set the condition flags.

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	0	0	1	0	0	0	0	0	1	Pg			0	Pn			M	Pd					
								B	S																						

BRKA <Pd>.B, <Pg>/<ZM>, <Pn>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	0	1	0	1	0	0	0	0	0	1		Pg		0			Pn		0			Pd		
								B	S															M							

BRKAS <Pd>.B, <Pg>/Z, <Pn>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<ZM> Is the predication qualifier, encoded in "M":

M	<ZM>
0	Z
1	M

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(PL) operand2 = P[d];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = if !break then '1' else '0';
        break = break || element;
    elseif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

BRKAS

Break after first true condition, setting the condition flags

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	1	0	1	0	0	0	0	0	1	Pg			0	Pn			0	Pd					
B										S											M										

BRKAS <Pd>.B, <Pg>/Z, <Pn>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(PL) operand2 = P[d];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = if !break then '1' else '0';
        break = break || element;
    elsif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BRKB, BRKBS

Break before first true condition

Not setting the condition flags

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Does not set the condition flags.

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	1	0	0	1	0	0	0	0	0	1	Pg			0	Pn			M	Pd					
								B	S																						

BRKB <Pd>.B, <Pg>/<ZM>, <Pn>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	1	1	0	1	0	0	0	0	0	1		Pg		0		Pn		0			Pd			
								B	S																M						

BRKBS <Pd>.B, <Pg>/Z, <Pn>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<ZM> Is the predication qualifier, encoded in "M":

M	<ZM>
0	Z
1	M

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(PL) operand2 = P[d];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if ElemP[mask, e, esize] == '1' then
        break = break || element;
        ElemP[result, e, esize] = if !break then '1' else '0';
    elseif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

BRKBS

Break before first true condition, setting the condition flags

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	1	1	0	1	0	0	0	0	0	1	Pg			0	Pn			0	Pd					
B										S										M											

BRKBS <Pd>.B, <Pg>/Z, <Pn>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(PL) operand2 = P[d];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if ElemP[mask, e, esize] == '1' then
        break = break || element;
        ElemP[result, e, esize] = if !break then '1' else '0';
    elsif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BRKN, BRKNS

Propagate break to next partition

Not setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Does not set the condition flags.

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	0	0	1	1	0	0	0	0	1	Pg			0	Pn			0	Pdm					
										S																					

BRKN <Pdm>.B, <Pg>/Z, <Pn>.B, <Pdm>.B

```
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	0	1	0	1	1	0	0	0	0	1	Pg			0	Pn			0	Pdm					
										S																					

BRKNS <Pdm>.B, <Pg>/Z, <Pn>.B, <Pdm>.B

```
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pdm> Is the name of the second source and destination scalable predicate register, encoded in the "Pdm" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[dm];
bits(PL) result;

if LastActive(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros();

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

BRKNS

Propagate break to next partition, setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	1	0	1	1	0	0	0	0	1		Pg		0			Pn		0		Pdm			

S

BRKNS <Pdm>.B, <Pg>/Z, <Pn>.B, <Pdm>.B

```
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pdm> Is the name of the second source and destination scalable predicate register, encoded in the "Pdm" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[dm];
bits(PL) result;

if LastActive(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros();

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm] = result;
```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BRKPA, BRKPAS

Break after first true condition, propagating from previous partition

Not setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	0	0	0		Pm			1	1		Pg		0			Pn		0			Pd		
S																B															

BRKPA <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	0	1	0	0		Pm			1	1		Pg		0			Pn		0			Pd		
S																B															

BRKPAS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
  if ElemP[mask, e, 8] == '1' then
    ElemP[result, e, 8] = if last then '1' else '0';
    last = last && (ElemP[operand2, e, 8] == '0');
  else
    ElemP[result, e, 8] = '0';

if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

BRKPAS

Break after first true condition, propagating from previous partition and setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	1	0	0	Pm			1		1	Pg			0		Pn			0		Pd			
S										B																					

BRKPAS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
    if ElemP[mask, e, 8] == '1' then
        ElemP[result, e, 8] = if last then '1' else '0';
        last = last && (ElemP[operand2, e, 8] == '0');
    else
        ElemP[result, e, 8] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BRKPB, BRKPBS

Break before first true condition, propagating from previous partition

Not setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	0	0	0				Pm		1	1			Pg		0			Pn		1			Pd
S																B															

BRKPB <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	0	1	0	0				Pm		1	1			Pg		0			Pn		1			Pd
S																B															

BRKPBS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
  if ElemP[mask, e, 8] == '1' then
    last = last && (ElemP[operand2, e, 8] == '0');
    ElemP[result, e, 8] = if last then '1' else '0';
  else
    ElemP[result, e, 8] = '0';

if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

BRKPBS

Break before first true condition, propagating from previous partition and setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	1	0	0	Pm			1		1	Pg			0		Pn			1		Pd			
S										B																					

BRKPBS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
    if ElemP[mask, e, 8] == '1' then
        last = last && (ElemP[operand2, e, 8] == '0');
        ElemP[result, e, 8] = if last then '1' else '0';
    else
        ElemP[result, e, 8] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR, EORS (predicates)

Bitwise exclusive OR predicates

Not setting the condition flags

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the [alias](#)~~aliases~~ [NOTS](#), and [NOT \(predicate\)](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	0	0	0	Pm				0	1	Pg				1	Pn				0	Pd			
S																															

EOR <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	0	1	0	0	Pm				0	1	Pg				1	Pn				0	Pd			
S																															

EORS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
NOT (predicate) NOTS	Pm == Pg

Alias	Is preferred when
<u>NOT (predicate)</u>	<u>Pm == Pg</u>

Operation

```

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 EOR element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

EORS

Bitwise exclusive OR predicates, setting the condition flags

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

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

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	1	0	0	Pm			0	1	Pg			1	Pn			0	Pd						
S																															

EORS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
NOTS	Pm == Pg

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 EOR element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

no old file

htmldiff from-

(new)

FMMLA

Floating-point matrix multiply-accumulate

The floating-point matrix multiply-accumulate instruction supports single-precision and double-precision data types in a 2×2 matrix contained in segments of 128 or 256 bits, respectively. It multiplies the 2×2 matrix in each segment of the first source vector by the 2×2 matrix in the corresponding segment of the second source vector. The resulting 2×2 matrix product is then destructively added to the matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing a 2-way dot product per destination element. This instruction is unpredicated. The single-precision variant is vector length agnostic. The double-precision variant requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F32MM indicates whether the single-precision variant is implemented.

ID_AA64ZFR0_EL1.F64MM indicates whether the double-precision variant is implemented.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

32-bit element

(FEAT_F32MM)

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	1	0	1	Zm					1	1	1	0	0	1	Zn					Zda				

FMMLA <Zda>.S, <Zn>.S, <Zm>.S

```
if !HaveSVEFP32MatMulExt() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

64-bit element

(FEAT_F64MM)

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	1	1	1	Zm					1	1	1	0	0	1	Zn					Zda				

FMMLA <Zda>.D, <Zn>.D, <Zm>.D

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
if VL < esize * 4 then UNDEFINED;
integer segments = VL DIV (4 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(4*esize) op1, op2;
bits(4*esize) res, addend;

for s = 0 to segments-1
    op1    = Elem[operand1, s, 4*esize];
    op2    = Elem[operand2, s, 4*esize];
    addend = Elem[operand3, s, 4*esize];
    res    = FPMatMulAdd(addend, op1, op2, esize, FPCR[]);
    Elem[result, s, 4*esize] = res;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+00:004122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1RB

Load and broadcast unsigned byte to vector

Load a single unsigned byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 4 classes: [8-bit element](#) , [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

8-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	1	imm6	1	0	0	Pg	Rn	Zt															
dtypeh<1>dtypeh<0>										dtypel<1>dtypel<0>																					

LD1RB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	1	imm6	1	0	1	Pg	Rn	Zt															
dtypeh<1>dtypeh<0>										dtypel<1>dtypel<0>																					

LD1RB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

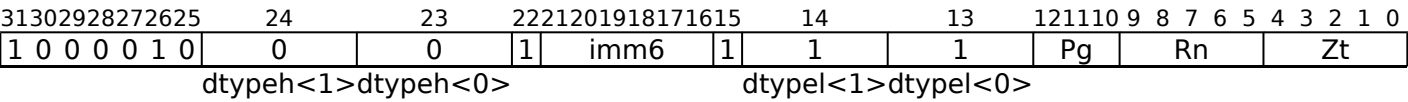
32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	1	imm6	1	1	0	Pg	Rn	Zt															
dtypeh<1>dtypeh<0>										dtypel<1>dtypel<0>																					

LD1RB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

64-bit element



LD1RB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1RH

Load and broadcast unsigned halfword to vector

Load a single unsigned halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	1	imm6						1	0	1	Pg			Rn			Zt						
dtypeh<1>dtypeh<0>																dtypel<1>dtypel<0>															

LD1RH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	1	imm6						1	1	0	Pg			Rn			Zt						
dtypeh<1>dtypeh<0>																dtypel<1>dtypel<0>															

LD1RH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	1	imm6						1	1	1	Pg			Rn			Zt						
dtypeh<1>dtypeh<0>																dtypel<1>dtypel<0>															

LD1RH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
```

Internal version only: isa v32.21v32.45, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1ROB (scalar plus immediate)

Contiguous load and replicate thirty-two bytes (immediate index)

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE

(FEAT_F64MM)

31	30	29	28	27	26	25	24	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	0	0	1	0	0	0	0	1	0	imm4			0	0	1	Pg			Rn			Zt							
msz<1>msz<0> ssz																															

LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1ROB (scalar plus scalar)

Contiguous load and replicate thirty-two bytes (scalar index)

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE

(FEAT_F64MM)

31	30	29	28	27	26	25	24	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	0	0	1	0	0	0	0	1	Rm				0	0	0	Pg			Rn				Zt									
msz<1>msz<0>							ssz																											

LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```

if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;

```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <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 offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1ROD (scalar plus immediate)

Contiguous load and replicate four doublewords (immediate index)

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE

(FEAT_F64MM)

31	30	29	28	27	26	25	24	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	0	0	1	0	1	1	0	1	0	imm4				0	0	1	Pg				Rn				Zt				
msz<1>msz<0>							ssz																								

LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1ROD (scalar plus scalar)

Contiguous load and replicate four doublewords (scalar index)

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE

(FEAT_F64MM)

31	30	29	28	27	26	25	24	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	0	0	1	0	1	1	0	1	Rm				0	0	0	Pg			Rn				Zt						
msz<1>msz<0>							ssz																								

LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```

if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;

```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <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 offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2; Build timestamp: 2021-06-28T16:20:41Z

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1ROH (scalar plus immediate)

Contiguous load and replicate sixteen halfwords (immediate index)

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE

(FEAT_F64MM)

31	30	29	28	27	26	25	24	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	0	0	1	0	0	1	0	1	0	imm4				0	0	1	Pg			Rn				Zt					
msz<1>msz<0>							ssz																								

LD1ROH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1ROH (scalar plus scalar)

Contiguous load and replicate sixteen halfwords (scalar index)

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE

(FEAT_F64MM)

31	30	29	28	27	26	25	24	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	0	0	1	0	0	1	0	1	Rm				0	0	0	Pg			Rn				Zt						
msz<1>msz<0>																ssz															

LD1ROH { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```

if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;

```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<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 offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2; Build timestamp: 2021-06-28T16:20:41Z

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1ROW (scalar plus immediate)

Contiguous load and replicate eight words (immediate index)

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE

(FEAT_F64MM)

31	30	29	28	27	26	25	24	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	0	0	1	0	1	0	0	1	0	imm4			0	0	1	Pg			Rn			Zt							
msz<1>							msz<0>							ssz																	

LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1ROW (scalar plus scalar)

Contiguous load and replicate eight words (scalar index)

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 4 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE

(FEAT_F64MM)

31	30	29	28	27	26	25	24	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	0	0	1	0	1	0	0	1	Rm			0	0	0	Pg			Rn			Zt								
msz<1>msz<0>							ssz																								

LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <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 offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2; Build timestamp: 2021-06-28T16:20:41Z

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1RSB

Load and broadcast signed byte to vector

Load a single signed byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	1	imm6	1	1	0	Pg	Rn	Zt																
dtypeh<1>dtypeh<0>										dtypel<1>dtypel<0>																					

LD1RSB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	1	imm6	1	0	1	Pg	Rn	Zt																
dtypeh<1>dtypeh<0>										dtypel<1>dtypel<0>																					

LD1RSB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	1	imm6	1	0	0	Pg	Rn	Zt																
dtypeh<1>dtypeh<0>										dtypel<1>dtypel<0>																					

LD1RSB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
```

Internal version only: isa v32.21v32.45, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1RSH

Load and broadcast signed halfword to vector

Load a single signed halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	0	1	imm6	1	0	1	Pg	Rn	Zt															
dtypeh<1>dtypeh<0>										dtypel<1>dtypel<0>																					

LD1RSH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	0	1	imm6	1	0	0	Pg	Rn	Zt															
dtypeh<1>dtypeh<0>										dtypel<1>dtypel<0>																					

LD1RSH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

LD1RSW

Load and broadcast signed word to vector

Load a single signed word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

31	30	29	28	27	26	25	24	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	0	0	1	0	0	1	1	imm6	1	0	0	Pg	Rn	Zt															
dtypeh<1>dtypeh<0>																dtypel<1>dtypel<0>															

LD1RSW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

LD1RW

Load and broadcast unsigned word to vector

Load a single unsigned word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	0	1	imm6	1	1	1	0	Pg	Rn	Zt														
dtypeh<1>dtypeh<0>										dtypel<1>dtypel<0>																					

LD1RW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	0	1	imm6	1	1	1	1	Pg	Rn	Zt														
dtypeh<1>dtypeh<0>										dtypel<1>dtypel<0>																					

LD1RW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

MOV (predicate, predicated, zeroing)

Move predicates (zeroing)

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of [AND, ANDS \(predicates\)](#). This means:

- The encodings in this description are named to match the encodings of [AND, ANDS \(predicates\)](#).
- The description of [AND, ANDS \(predicates\)](#) gives the operational pseudocode for this instruction.

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

MOV <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to

AND <Pd>.B, <Pg>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when S == '0' && Pn == Pm.

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of [AND, ANDS \(predicates\)](#) gives the operational pseudocode for this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (SIMD&FP scalar, unpredicated)

Move indexed element or SIMD&FP scalar to vector (unpredicated)

Unconditionally broadcast the SIMD&FP scalar into each element of the destination vector. This instruction is unpredicated.

This is an alias of [DUP \(indexed\)](#). This means:

- The encodings in this description are named to match the encodings of [DUP \(indexed\)](#).
- The description of [DUP \(indexed\)](#) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	imm2		1	tsz					0	0	1	0	0	0	Zn				Zd					

MOV <Zd>.<T>, <Zn><V><n>.<T>[<imm>]

is equivalent to

DUP <Zd>.<T>, <Zn>.<T>[0]<imm>]

and is the preferred disassembly when BitCount(imm2:tsz) >= 1.

MOV <Zd>.<T>, <V><Zn>.<T>[<imm>]<n>

is equivalent to

DUP <Zd>.<T>, <Zn>.<T>[0]<imm>]

and is the preferred disassembly when BitCount(imm2:tsz) == 1.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsz":

tsz	<T>
00000	RESERVED
xxx1	B
xx10	H
xx100	S
x1000	D
10000	Q

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm> Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in "imm2:tsz".

<V> Is a width specifier, encoded in "tsz":

tsz	<V>
00000	RESERVED
xxx1	B
xx10	H
xx100	S
x1000	D
10000	Q

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the "Zn" field.

Operation

The description of [DUP \(indexed\)](#) gives the operational pseudocode for this instruction.

Internal version only: isa ~~v32.21~~v32.45, AdvSIMD v29.05, pseudocode ~~v2021-06_xml~~v2021-03, sve ~~v2021-06_rc2~~v2021-03-rc2 ; Build timestamp: ~~2021-06-28T16:41:22~~2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

MOV (predicate, unpredicated)

Move predicate (unpredicated)

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Does not set the condition flags.

This is an alias of [ORR, ORRS \(predicates\)](#). This means:

- The encodings in this description are named to match the encodings of [ORR, ORRS \(predicates\)](#).
- The description of [ORR, ORRS \(predicates\)](#) gives the operational pseudocode for this instruction.

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

MOV <Pd>.B, <Pn>.B

is equivalent to

ORR <Pd>.B, <Pn>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when `S == '0' && Pn == Pm && Pm == Pg`.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of [ORR, ORRS \(predicates\)](#) gives the operational pseudocode for this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bvv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOVS (predicated)

Move predicates (zeroing), setting the condition flags

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of [ANDS](#). This means:

- The encodings in this description are named to match the encodings of [ANDS](#).
- The description of [ANDS](#) gives the operational pseudocode for this instruction.

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

MOVS <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to

ANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when **S == '1' && Pn == Pm**.

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of [ANDS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOVS (unpredicated)

Move predicate (unpredicated), setting the condition flags

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of [ORRS](#). This means:

- The encodings in this description are named to match the encodings of [ORRS](#).
- The description of [ORRS](#) gives the operational pseudocode for this instruction.

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

MOVS <Pd>.B, <Pn>.B

is equivalent to

ORRS <Pd>.B, <Pn>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when $S == '1' \ \&\& \ Pn == Pm \ \&\& \ Pm == Pg$.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of [ORRS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NAND, NANDS

Bitwise NAND predicates

Not setting the condition flags

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	1	0	0	0	Pm			0		1	Pg			1		Pn			1		Pd			
S																															

NAND <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	1	1	0	0		Pm			0	1		Pg			1			Pn			1			Pd

S

NANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 AND element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+01:00

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

NANDS

Bitwise NAND predicates, setting the condition flags

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	1	1	0	0	Pm			0		1	Pg			1		Pn			1		Pd			
																	S														

NANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 AND element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NOR, NORS

Bitwise NOR predicates

Not setting the condition flags

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	1	0	0	0	Pm			0		1	Pg			1		Pn			0		Pd			
S																															

NOR <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	1	1	0	0	Pm			0	1	Pg			1	Pn			0	Pd						
S																															

NORS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 OR element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+01:00

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

NORS

Bitwise NOR predicates, setting the condition flags

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	1	1	0	0		Pm			0	1		Pg		1		Pn			0			Pd		
S																															

NORS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 OR element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

Bitwise invert predicate

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of EOR, EORS (predicates). This means:

- The encodings in this description are named to match the encodings of [EOR, EORS \(predicates\)](#).
- The description of [EOR, EORS \(predicates\)](#) gives the operational pseudocode for this instruction.

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

S

NOT $\langle Pd \rangle .B$, $\langle Pg \rangle /Z$, $\langle Pn \rangle .B$

is equivalent to

EOR **<Pd>.B**, **<Pg>/Z**, **<Pn>.B**, **<Pg>.B**

and is the preferred disassembly when $P_m == P_g$.

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of EOR, EORS (predicates) gives the operational pseudocode for this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NOTS

Bitwise invert predicate, setting the condition flags

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of [EORS](#). This means:

- The encodings in this description are named to match the encodings of [EORS](#).
- The description of [EORS](#) gives the operational pseudocode for this instruction.

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

NOTS <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to

[EORS](#) <Pd>.B, <Pg>/Z, <Pn>.B, <Pg>.B

and is the preferred disassembly when Pm == Pg.

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of [EORS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORN, ORNS (predicates)

Bitwise inclusive OR inverted predicate

Not setting the condition flags

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	1	0	0	0	Pm			0	1	Pg			0	Pn			1	Pd						
S																															

ORN <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	1	1	0	0		Pm			0	1		Pg			0			Pn			1			Pd

S

ORNS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21+01:004122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

ORNS

Bitwise inclusive OR inverted predicate, setting the condition flags

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	1	1	0	0	Pm			0		1	Pg			0		Pn			1		Pd				
S																																

ORNS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

Not setting the condition flags**ORR, ORRS (predicates)**

Bitwise inclusive OR ~~predicates~~predicate

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

~~Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.~~

This instruction is used by the ~~alias~~aliases ~~MOVS (unpredicated)~~, and ~~MOV (predicate, unpredicated)~~.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	1	0	0	0	Pm			0	1	Pg			0	Pn			0	Pd						
S																															

ORR <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	1	1	0	0		Pm			0	1		Pg			0		Pn			0		Pd		
S																															

ORRS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
MOV (predicate, unpredicated) MOVVS (unpredicated)	S == '01' && Pn == Pm && Pm == Pg

Alias	Is preferred when
MOV (predicate, unpredicated)	$S == '0' \ \&\& \ P_n == P_m \ \&\& \ P_m == P_g$

Operation

```

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

```

Internal version only: isa [v32.21](#)[v32.15](#), AdvSIMD v29.05, pseudocode [v2021-06_xml](#)[v2021-03](#), sve [v2021-06_rc2](#)[v2021-03_rc2](#) ; Build timestamp: [2021-06-28T16:41:22](#)[2021-03-30T21:41:22](#)

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

ORRS

Bitwise inclusive OR predicates, setting the condition flags

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias [MOVS \(unpredicated\)](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	1	1	0	0		Pm			0	1		Pg			0		Pn			0		Pd		
S																															

ORRS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
MOVS (unpredicated)	<code>S == '1' && Pn == Pm && Pm == Pg</code>

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```


no old file

htmldiff from-

(new)

PTRUE, PTRUES

Initialise predicate from named constraint

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:

- * A fixed number (VL1 to VL256)
- * The largest power of two (POW2)
- * The largest multiple of three or four (MUL3 or MUL4)
- * All available, implicitly a multiple of two (ALL).

Not setting the condition flags

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Does not set the condition flags.

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size	0	1	1	0	0	0	0	1	1	1	0	0	0	pattern					0	Pd			
																S															

PTRUE <Pd>.<T>{, <pattern>}

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = FALSE;
bits(5) pat = pattern;
```

Setting the 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
0	0	1	0	0	1	0	1	size	0	1	1	0	0	1	1	1	1	0	0	0	pattern					0	Pd				
																S															

~~PTRUES <Pd>.<T>{, <pattern>}~~

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = TRUE;
bits(5) pat = pattern;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	P0W2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Operation

```

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;

for e = 0 to elements-1
    ElemP[result, e, esize] = if e < count then '1' else '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d] = result;

```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:21.4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

PTRUES

Initialise predicate from named constraint and set the condition flags

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:

- * A fixed number (VL1 to VL256)
- * The largest power of two (POW2)
- * The largest multiple of three or four (MUL3 or MUL4)
- * All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size	0	1	1	0	0	1	1	1	1	0	0	0	pattern	0					0			Pd	

S

PTRUES <Pd>.<T>{, <pattern>}

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = TRUE;
bits(5) pat = pattern;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Operation

```

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;

for e = 0 to elements-1
    ElemP[result, e, esize] = if e < count then '1' else '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d] = result;

```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

no old file

htmldiff from-

(new)

RDFFR, RDFFRS (predicated)

Return predicate of successfully loaded elements

Not setting the condition flags

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	0	0	1	1	0	0	0	1	1	1	1	0	0	0			Pg		0			Pd	
S																															

RDFFR <Pd>.B, <Pg>/Z

```
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Setting the 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
0	0	1	0	0	1	0	1	0	1	0	1	1	0	0	0	1	1	1	1	0	0	0			Pg		0			Pd	
S																															

RDFFRS <Pd>.B, <Pg>/Z

```
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

```
CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) ffr = FFR[];
bits(PL) result = ffr AND mask;

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d] = result;
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RDFFRS

Return predicate of successfully loaded elements, setting the condition flags

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	1	0	1	1	0	0	0	1	1	1	1	0	0	0				Pg		0			Pd
S																															

RDFFRS <Pd>.B, <Pg>/Z

```
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

```
CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) ffr = FFR[];
bits(PL) result = ffr AND mask;

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d] = result;
```

Internal version only: isa v32.21, AdvSIMD v29.05, pseudocode v2021-06_xml, sve v2021-06_rc2b ; Build timestamp: 2021-06-28T16:41

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMMLA

Signed integer matrix multiply-accumulate

The signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of signed 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE

(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	0	1	0	1	0	0	0	Zm				1	0	0	1	1	0	Zn				Zda						
uns<1>								uns<0>																							

SMMLA <Zda>.S, <Zn>.B, <Zm>.B

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

SUDOT

Signed by unsigned integer indexed dot product

The signed by unsigned integer indexed dot product instruction computes the dot product of a group of four signed 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four unsigned 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE

(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	0	1	0	0	1	0	1	i2	Zm	0	0	0	1	1	1	Zn					Zda							
size<1>size<0>								U																							

SUDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>]

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index of a quadruplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltsegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltsegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2hv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

TRN1, TRN2 (vectors)

Interleave even or odd elements from two vectors

Interleave alternating even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.

It has encodings from 4 classes: [Even](#) , [Even \(quadwords\)](#) , [Odd](#) and [Odd \(quadwords\)](#)

Even

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

H

TRN1 [<Zd>.<T>](#), [<Zn>.<T>](#), [<Zm>.<T>](#)

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Even (quadwords)

(FEAT_F64MM)

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

H

TRN1 [<Zd>.Q](#), [<Zn>.Q](#), [<Zm>.Q](#)

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Odd

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

H

TRN2 [<Zd>.<T>](#), [<Zn>.<T>](#), [<Zm>.<T>](#)

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

(FEAT_F64MM)

H

```

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

```

UMMLA

Unsigned integer matrix multiply-accumulate

The unsigned integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of unsigned 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE

(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	0	1	0	1	1	1	0			Zm			1	0	0	1	1	0			Zn					Zda		

uns<1>uns<0>

UMMLA <Zda>.S, <Zn>.B, <Zm>.B

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

USDOT (vectors)

Unsigned by signed integer dot product

The unsigned by signed integer dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE

(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	0	1	0	0	1	0	0	Zm				0	1	1	1	1	0	Zn				Zda						

size<1>size<0>

USDOT <Zda>.S, <Zn>.B, <Zm>.B

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03-re2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

USDOT (indexed)

Unsigned by signed integer indexed dot product

The unsigned by signed integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE

(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	0	1	0	0	1	0	1	i2	Zm	0	0	0	1	1	0	Zn					Zda							
size<1>size<0>								U																							

USDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>]

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index of a quadruplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltsegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltsegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

USMMLA

Unsigned by signed integer matrix multiply-accumulate

The unsigned by signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE

(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	0	1	0	1	1	0	0			Zm			1	0	0	1	1	0			Zn					Zda		
								uns<1>		uns<0>																					

USMMLA <Zda>.S, <Zn>.B, <Zm>.B

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

UZP1, UZP2 (vectors)

Concatenate even or odd elements from two vectors

Concatenate adjacent even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.

It has encodings from 4 classes: [Even](#) , [Even \(quadwords\)](#) , [Odd](#) and [Odd \(quadwords\)](#)

Even

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

H

UZP1 [<Zd>.<T>](#), [<Zn>.<T>](#), [<Zm>.<T>](#)

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Even (quadwords)

(FEAT_F64MM)

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

UZP1 [<Zd>.Q](#), [<Zn>.Q](#), [<Zm>.Q](#)

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Odd

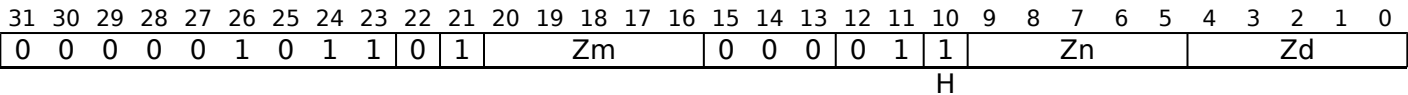
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	Zm			0	1	1	0	1	1	Zn			Zd									

H

UZP2 [<Zd>.<T>](#), [<Zn>.<T>](#), [<Zm>.<T>](#)

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Odd (quadwords)
(FEAT_F64MM)



UZP2 <Zd>.Q, <Zn>.Q, <Zm>.Q

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
if VL < esize * 2 then UNDEFINED;
integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Zeros();

for p = 0 to pairs - 1
    Elem[result, p, esize] = Elem[operand1, 2*p+part, esize];

for p = 0 to pairs - 1
    Elem[result, pairs+p, esize] = Elem[operand2, 2*p+part, esize];

Z[d] = result;
```

Internal version only: isa v32.21v32.45, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:2021-03-30T21:4122

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

ZIP1, ZIP2 (vectors)

Interleave elements from two half vectors

Interleave alternating elements from the lowest or highest halves of the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.

It has encodings from 4 classes: [High halves](#) , [High halves \(quadwords\)](#) , [Low halves](#) and [Low halves \(quadwords\)](#)

High halves

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

H

ZIP2 [<Zd>.<T>](#), [<Zn>.<T>](#), [<Zm>.<T>](#)

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

High halves (quadwords)

(FEAT_F64MM)

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

H

ZIP2 [<Zd>.Q](#), [<Zn>.Q](#), [<Zm>.Q](#)

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Low halves

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

H

ZIP1 [<Zd>.<T>](#), [<Zn>.<T>](#), [<Zm>.<T>](#)

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```


(FEAT F64MM)

H

```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

(old)

htmldiff from-

(new)

Top-level encodings for A64

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0																															

Decode fields		Instruction details
op0		
0000		Reserved
0001		UNALLOCATED
0010		SVE encodings
0011		UNALLOCATED
100x		Data Processing -- Immediate
101x		Branches, Exception Generating and System instructions
x1x0		Loads and Stores
x101		Data Processing -- Register
x111		Data Processing -- Scalar Floating-Point and Advanced SIMD

Reserved

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0						0000						op1																			

Decode fields		Instruction details
op0	op1	
000	000000000	UDF
	!= 000000000	UNALLOCATED
!= 000		UNALLOCATED

SVE encodings

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0						0010						op1				op2				op3											

Decode fields				Instruction details
op0	op1	op2	op3	
000	0x	0xxxx	x1xxxx	SVE Integer Multiply-Add - Predicated
000	0x	0xxxx	000xxx	SVE Integer Binary Arithmetic - Predicated
000	0x	0xxxx	001xxx	SVE Integer Reduction
000	0x	0xxxx	100xxx	SVE Bitwise Shift - Predicated
000	0x	0xxxx	101xxx	SVE Integer Unary Arithmetic - Predicated
000	0x	1xxxx	000xxx	SVE integer add/subtract vectors (unpredicated)
000	0x	1xxxx	001xxx	SVE Bitwise Logical - Unpredicated
000	0x	1xxxx	0100xx	SVE Index Generation
000	0x	1xxxx	0101xx	SVE Stack Allocation
000	0x	1xxxx	011xxx	UNALLOCATED
000	0x	1xxxx	100xxx	SVE Bitwise Shift - Unpredicated

000	0x	1xxxx	1010xx	SVE address generation
000	0x	1xxxx	1011xx	SVE Integer Misc - Unpredicated
000	0x	1xxxx	11xxxx	SVE Element Count
000	1x	00xxx		SVE Bitwise Immediate
000	1x	01xxx		SVE Integer Wide Immediate - Predicated
000	1x	1xxxx	001000	DUP (indexed)
000	1x	1xxxx	001001	UNALLOCATED
000	1x	1xxxx	00101x	UNALLOCATED
000	1x	1xxxx	0011x1	UNALLOCATED
000	1x	1xxxx	001100	TBL
000	1x	1xxxx	001110	SVE Permute Vector - Unpredicated
000	1x	1xxxx	010xxx	SVE Permute Predicate
000	1x	1xxxx	011xxx	SVE permute vector elements
000	1x	1xxxx	10xxxx	SVE Permute Vector - Predicated
000	1x	1xxxx	11xxxx	SEL (vectors)
000	10	1xxxx	000xxx	SVE Permute Vector - Extract
000	11	1xxxx	000xxx	SVE permute vector segments
001	0x	0xxxx		SVE Integer Compare - Vectors
001	0x	1xxxx		SVE integer compare with unsigned immediate
001	1x	0xxxx	x0xxxx	SVE integer compare with signed immediate
001	1x	00xxx	01xxxx	SVE predicate logical operations
001	1x	00xxx	11xxxx	SVE Propagate Break
001	1x	01xxx	01xxxx	SVE Partition Break
001	1x	01xxx	11xxxx	SVE Predicate Misc
001	1x	1xxxx	00xxxx	SVE Integer Compare - Scalars
001	1x	1xxxx	01xxxx	UNALLOCATED
001	1x	1xxxx	11xxxx	SVE Integer Wide Immediate - Unpredicated
001	1x	100xx	10xxxx	SVE Predicate Count
001	1x	101xx	1000xx	SVE Inc/Dec by Predicate Count
001	1x	101xx	1001xx	SVE Write FFR
001	1x	101xx	101xxx	UNALLOCATED
001	1x	11xxx	10xxxx	UNALLOCATED
010	0x	0xxxx	0xxxxxx	SVE Integer Multiply-Add - Unpredicated
010	0x	0xxxx	1xxxxxx	UNALLOCATED
010	0x	1xxxx		SVE Multiply - Indexed
010	1x	0xxxx	0xxxxxx	UNALLOCATED
010	1x	0xxxx	10xxxx	SVE Misc
010	1x	0xxxx	11xxxx	UNALLOCATED
010	1x	1xxxx		UNALLOCATED
011	0x	0xxxx	0xxxxxx	FCMLA (vectors)
011	0x	00x1x	1xxxxxx	UNALLOCATED
011	0x	00000	100xxx	FCADD
011	0x	00000	101xxx	UNALLOCATED
011	0x	00000	11xxxx	UNALLOCATED
011	0x	00001	1xxxxxx	UNALLOCATED
011	0x	0010x	100xxx	UNALLOCATED
011	0x	0010x	101xxx	SVE floating-point convert precision odd elements
011	0x	0010x	11xxxx	UNALLOCATED
011	0x	01xxx	1xxxxxx	UNALLOCATED

011	0x	1xxxx	x0x01x	UNALLOCATED
011	0x	1xxxx	00000x	SVE floating-point multiply-add (indexed)
011	0x	1xxxx	0001xx	SVE floating-point complex multiply-add (indexed)
011	0x	1xxxx	001000	SVE floating-point multiply (indexed)
011	0x	1xxxx	001001	UNALLOCATED
011	0x	1xxxx	0011xx	UNALLOCATED
011	0x	1xxxx	01x0xx	SVE Floating Point Widening Multiply-Add - Indexed
011	0x	1xxxx	01x1xx	UNALLOCATED
011	0x	1xxxx	10x00x	SVE Floating Point Widening Multiply-Add
011	0x	1xxxx	10x1xx	UNALLOCATED
011	0x	1xxxx	110xxx	UNALLOCATED
011	0x	1xxxx	111000	UNALLOCATED
011	0x	1xxxx	111001	SVE floating point matrix multiply accumulate
011	0x	1xxxx	11101x	UNALLOCATED
011	0x	1xxxx	1111xx	UNALLOCATED
011	1x	0xxxx	x1xxxx	SVE floating-point compare vectors
011	1x	0xxxx	000xxx	SVE floating-point arithmetic (unpredicated)
011	1x	0xxxx	100xxx	SVE Floating Point Arithmetic - Predicated
011	1x	0xxxx	101xxx	SVE Floating Point Unary Operations - Predicated
011	1x	000xx	001xxx	SVE floating-point recursive reduction
011	1x	001xx	0010xx	UNALLOCATED
011	1x	001xx	0011xx	SVE Floating Point Unary Operations - Unpredicated
011	1x	010xx	001xxx	SVE Floating Point Compare - with Zero
011	1x	011xx	001xxx	SVE Floating Point Accumulating Reduction
011	1x	1xxxx		SVE Floating Point Multiply-Add
100				SVE Memory - 32-bit Gather and Unsized Contiguous
101				SVE Memory - Contiguous Load
110				SVE Memory - 64-bit Gather
111			0x0xxx	SVE Memory - Contiguous Store and Unsized Contiguous
111			0x1xxx	SVE Memory - Non-temporal and Multi-register Store
111			1x0xxx	SVE Memory - Scatter with Optional Sign Extend
111			101xxx	SVE Memory - Scatter
111			111xxx	SVE Memory - Contiguous Store with Immediate Offset

SVE Integer Multiply-Add - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100									0	op0						1															

Decode fields
op0

Instruction details

0	SVE integer multiply-accumulate writing addend (predicated)
1	SVE integer multiply-add writing multiplicand (predicated)

SVE integer multiply-accumulate writing addend (predicated)

These instructions are under [SVE Integer Multiply-Add - Predicated](#).

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

Decode fields op	Instruction Details
0	MLA
1	MLS

SVE integer multiply-add writing multiplicand (predicated)

These instructions are under [SVE Integer Multiply-Add - Predicated](#).

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

Decode fields op	Instruction Details
0	MAD
1	MSB

SVE Integer Binary Arithmetic - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										0		op0							000												

Decode fields op0	Instruction details
00x	SVE integer add/subtract vectors (predicated)
01x	SVE integer min/max/difference (predicated)
100	SVE integer multiply vectors (predicated)
101	SVE integer divide vectors (predicated)
11x	SVE bitwise logical operations (predicated)

SVE integer add/subtract vectors (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

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

Decode fields opc	Instruction Details
000	ADD (vectors, predicated)
001	SUB (vectors, predicated)
010	UNALLOCATED
011	SUBR (vectors)
1xx	UNALLOCATED

SVE integer min/max/difference (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

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

Decode fields opc	U	Instruction Details
00	0	SMAX (vectors)
00	1	UMAX (vectors)
01	0	SMIN (vectors)
01	1	UMIN (vectors)
10	0	SABD
10	1	UABD
11		UNALLOCATED

SVE integer multiply vectors (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

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

Decode fields H	U	Instruction Details
0	0	MUL (vectors)
0	1	UNALLOCATED
1	0	SMULH
1	1	UMULH

SVE integer divide vectors (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

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

Decode fields R	U	Instruction Details
0	0	SDIV
0	1	UDIV
1	0	SDIVR
1	1	UDIVR

SVE bitwise logical operations (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

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

Decode fields opc	Instruction Details
000	ORR (vectors, predicated)
001	EOR (vectors, predicated)
010	AND (vectors, predicated)
011	BIC (vectors, predicated)
1xx	UNALLOCATED

SVE Integer Reduction

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100									0	op0						001															

Decode fields op0	Instruction details
000	SVE integer add reduction (predicated)
010	SVE integer min/max reduction (predicated)
0x1	UNALLOCATED
10x	SVE constructive prefix (predicated)
110	SVE bitwise logical reduction (predicated)
111	UNALLOCATED

SVE integer add reduction (predicated)

These instructions are under [SVE Integer Reduction](#).

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

Decode fields op	U	Instruction Details
0	0	SADDV
0	1	UADDV
1		UNALLOCATED

SVE integer min/max reduction (predicated)

These instructions are under [SVE Integer Reduction](#).

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

Decode fields op	U	Instruction Details
0	0	SMAV
0	1	UMAV
1	0	SMINV
1	1	UMINV

SVE constructive prefix (predicated)

These instructions are under [SVE Integer Reduction](#).

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

Decode fields opc	Instruction Details
00	MOVPRFX (predicated)
01	UNALLOCATED
1x	UNALLOCATED

SVE bitwise logical reduction (predicated)

These instructions are under [SVE Integer Reduction](#).

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

Decode fields opc	Instruction Details
00	ORV
01	EORV
10	ANDV
11	UNALLOCATED

SVE Bitwise Shift - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					00000100				0	op0						100															

Decode fields op0	Instruction details
0x	SVE bitwise shift by immediate (predicated)
10	SVE bitwise shift by vector (predicated)
11	SVE bitwise shift by wide elements (predicated)

SVE bitwise shift by immediate (predicated)

These instructions are under [SVE Bitwise Shift - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	tszh	0	0	opc	L	U	1	0	0			Pg			tszl		imm3					Zdn		

Decode fields opc	L	U	Instruction Details
00	0	0	ASR (immediate, predicated)
00	0	1	LSR (immediate, predicated)
00	1	0	UNALLOCATED
00	1	1	LSL (immediate, predicated)
01	0	0	ASRD
01	0	1	UNALLOCATED
01	1		UNALLOCATED
1x			UNALLOCATED

SVE bitwise shift by vector (predicated)

These instructions are under [SVE Bitwise Shift - Predicated](#).

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

Decode fields R	L	U	Instruction Details
	1	0	UNALLOCATED
0	0	0	ASR (vectors)

Decode fields			Instruction Details
R	L	U	
0	0	1	LSR (vectors)
0	1	1	LSL (vectors)
1	0	0	ASRR
1	0	1	LSRR
1	1	1	LSLR

SVE bitwise shift by wide elements (predicated)

These instructions are under [SVE Bitwise Shift - Predicated](#).

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

Decode fields			Instruction Details
R	L	U	
0	0	0	ASR (wide elements, predicated)
0	0	1	LSR (wide elements, predicated)
0	1	0	UNALLOCATED
0	1	1	LSL (wide elements, predicated)
1			UNALLOCATED

SVE Integer Unary Arithmetic - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100										0	op0					101															

Decode fields		Instruction details
op0		
0x		UNALLOCATED
10		SVE integer unary operations (predicated)
11		SVE bitwise unary operations (predicated)

SVE integer unary operations (predicated)

These instructions are under [SVE Integer Unary Arithmetic - Predicated](#).

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

Decode fields	Instruction Details
opc	
000	SXTB, SXTH, SXTW — SXTB
001	UXTB, UXTH, UXTW — UXTB
010	SXTB, SXTH, SXTW — SXTH
011	UXTB, UXTH, UXTW — UXTH
100	SXTB, SXTH, SXTW — SXTW
101	UXTB, UXTH, UXTW — UXTW
110	ABS
111	NEG

SVE bitwise unary operations (predicated)

These instructions are under [SVE Integer Unary Arithmetic - Predicated](#).

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

Decode fields opc	Instruction Details
000	CLS
001	CLZ
010	CNT
011	CNOT
100	FABS
101	FNEG
110	NOT (vector)
111	UNALLOCATED

SVE integer add/subtract vectors (unpredicated)

These instructions are under [SVE encodings](#).

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

Decode fields opc	Instruction Details
000	ADD (vectors, unpredicated)
001	SUB (vectors, unpredicated)
01x	UNALLOCATED
100	SQADD (vectors)
101	UQADD (vectors)
110	SQSUB (vectors)
111	UQSUB (vectors)

SVE Bitwise Logical - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
00000100									1					001	op0	op1																		

Decode fields op0	op1	Instruction details
0		UNALLOCATED
1	00	SVE bitwise logical operations (unpredicated)
1	!= 00	UNALLOCATED

SVE bitwise logical operations (unpredicated)

These instructions are under [SVE Bitwise Logical - Unpredicated](#).

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

Decode fields opc	Instruction Details
00	AND (vectors, unpredicated)
01	ORR (vectors, unpredicated)
10	EOR (vectors, unpredicated)
11	BIC (vectors, unpredicated)

SVE Index Generation

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100									1							0100			op0												

Decode fields op0	Instruction details
00	INDEX (immediates)
01	INDEX (scalar, immediate)
10	INDEX (immediate, scalar)
11	INDEX (scalars)

SVE Stack Allocation

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
00000100								op0			1							0101			op1											

Decode fields op0	Decode fields op1	Instruction details
0	0	SVE stack frame adjustment
1	0	SVE stack frame size
	1	UNALLOCATED

SVE stack frame adjustment

These instructions are under [SVE Stack Allocation](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	op		1	Rn						0 1 0 1 0				imm6						Rd			

Decode fields op	Instruction Details
0	ADDVL
1	ADDPL

SVE stack frame size

These instructions are under [SVE Stack Allocation](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	op		1	opc2						0 1 0 1 0				imm6						Rd			

Decode fields op	opc2	Instruction Details
0	0XXXX	UNALLOCATED
0	10XXX	UNALLOCATED
0	110XX	UNALLOCATED
0	1110x	UNALLOCATED
0	11110	UNALLOCATED
0	11111	RDVL
1		UNALLOCATED

SVE Bitwise Shift - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100									1							100	op0														

Decode fields op0	Instruction details
0	SVE bitwise shift by wide elements (unpredicated)
1	SVE bitwise shift by immediate (unpredicated)

SVE bitwise shift by wide elements (unpredicated)

These instructions are under [SVE Bitwise Shift - Unpredicated](#).

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

Decode fields opc	Instruction Details
00	ASR (wide elements, unpredicated)
01	LSR (wide elements, unpredicated)
10	UNALLOCATED
11	LSL (wide elements, unpredicated)

SVE bitwise shift by immediate (unpredicated)

These instructions are under [SVE Bitwise Shift - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	tszh	1	tszl	imm3			1	0	0	1	opc	Zn						Zd						

Decode fields opc	Instruction Details
00	ASR (immediate, unpredicated)
01	LSR (immediate, unpredicated)
10	UNALLOCATED
11	LSL (immediate, unpredicated)

SVE address generation

These instructions are under [SVE encodings](#).

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

Decode fields	Instruction Details
opc	
00	ADR — Unpacked 32-bit signed offsets
01	ADR — Unpacked 32-bit unsigned offsets
1x	ADR — Packed offsets

SVE Integer Misc - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100									1							1011	op0														

Decode fields	Instruction details
op0	
0x	SVE floating-point trig select coefficient
10	SVE floating-point exponential accelerator
11	SVE constructive prefix (unpredicated)

SVE floating-point trig select coefficient

These instructions are under [SVE Integer Misc - Unpredicated](#).

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

Decode fields	Instruction Details
op	
0	FTSSEL
1	UNALLOCATED

SVE floating-point exponential accelerator

These instructions are under [SVE Integer Misc - Unpredicated](#).

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

Decode fields	Instruction Details
opc	
00000	FEXPA
00001	UNALLOCATED
0001x	UNALLOCATED
001xx	UNALLOCATED
01xxx	UNALLOCATED
1xxxx	UNALLOCATED

SVE constructive prefix (unpredicated)

These instructions are under [SVE Integer Misc - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	opc	1	opc2						1	0	1	1	1	1	Zn						Zd			

Decode fields		Instruction Details
opc	opc2	
00	00000	MOVPRFX (unpredicated)
00	00001	UNALLOCATED
00	0001x	UNALLOCATED
00	001xx	UNALLOCATED
00	01xxx	UNALLOCATED
00	1xxxx	UNALLOCATED
01		UNALLOCATED
1x		UNALLOCATED

SVE Element Count

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100										1	op0					11	op1														

Decode fields		Instruction details
op0	op1	
0	00x	SVE saturating inc/dec vector by element count
0	100	SVE element count
0	101	UNALLOCATED
1	000	SVE inc/dec vector by element count
1	100	SVE inc/dec register by element count
1	x01	UNALLOCATED
	01x	UNALLOCATED
	11x	SVE saturating inc/dec register by element count

SVE saturating inc/dec vector by element count

These instructions are under [SVE Element Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		1	0	imm4				1	1	0	0	D	U	pattern				Zdn					

Decode fields			Instruction Details
size	D	U	
00			UNALLOCATED
01	0	0	SQINCH (vector)
01	0	1	UQINCH (vector)
01	1	0	SQDECH (vector)
01	1	1	UQDECH (vector)
10	0	0	SQINCW (vector)
10	0	1	UQINCW (vector)
10	1	0	SQDECW (vector)
10	1	1	UQDECW (vector)
11	0	0	SQINCD (vector)
11	0	1	UQINCD (vector)
11	1	0	SQDECD (vector)
11	1	1	UQDECD (vector)

SVE element count

These instructions are under [SVE Element Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	0		imm4	1	1	1	0	0	op		pattern											Rd

Decode fields size	op	Instruction Details
	1	UNALLOCATED
00	0	CNTB, CNTD, CNTH, CNTW — CNTB
01	0	CNTB, CNTD, CNTH, CNTW — CNTH
10	0	CNTB, CNTD, CNTH, CNTW — CNTW
11	0	CNTB, CNTD, CNTH, CNTW — CNTD

SVE inc/dec vector by element count

These instructions are under [SVE Element Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	1		imm4	1	1	0	0	0	D		pattern											Zdn

Decode fields size	D	Instruction Details
00		UNALLOCATED
01	0	INCD, INCH, INCW (vector) — INCH
01	1	DECD, DECH, DECW (vector) — DECH
10	0	INCD, INCH, INCW (vector) — INCW
10	1	DECD, DECH, DECW (vector) — DECW
11	0	INCD, INCH, INCW (vector) — INCD
11	1	DECD, DECH, DECW (vector) — DECD

SVE inc/dec register by element count

These instructions are under [SVE Element Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	1		imm4	1	1	1	0	0	D		pattern											Rdn

Decode fields size	D	Instruction Details
00	0	INCB, INCD, INCH, INCW (scalar) — INCB
00	1	DECB, DECD, DECH, DECW (scalar) — DECB
01	0	INCB, INCD, INCH, INCW (scalar) — INCH
01	1	DECB, DECD, DECH, DECW (scalar) — DECH
10	0	INCB, INCD, INCH, INCW (scalar) — INCW
10	1	DECB, DECD, DECH, DECW (scalar) — DECW
11	0	INCB, INCD, INCH, INCW (scalar) — INCD
11	1	DECB, DECD, DECH, DECW (scalar) — DECD

SVE saturating inc/dec register by element count

These instructions are under [SVE Element Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	sf		imm4	1	1	1	1	D	U		pattern											Rdn

Decode fields				Instruction Details
size	sf	D	U	
00	0	0	0	SQINCB — 32-bit
00	0	0	1	UQINCB — 32-bit
00	0	1	0	SQDECB — 32-bit
00	0	1	1	UQDECB — 32-bit
00	1	0	0	SQINCB — 64-bit
00	1	0	1	UQINCB — 64-bit
00	1	1	0	SQDECB — 64-bit
00	1	1	1	UQDECB — 64-bit
01	0	0	0	SQINCH (scalar) — 32-bit
01	0	0	1	UQINCH (scalar) — 32-bit
01	0	1	0	SQDECH (scalar) — 32-bit
01	0	1	1	UQDECH (scalar) — 32-bit
01	1	0	0	SQINCH (scalar) — 64-bit
01	1	0	1	UQINCH (scalar) — 64-bit
01	1	1	0	SQDECH (scalar) — 64-bit
01	1	1	1	UQDECH (scalar) — 64-bit
10	0	0	0	SQINCW (scalar) — 32-bit
10	0	0	1	UQINCW (scalar) — 32-bit
10	0	1	0	SQDECW (scalar) — 32-bit
10	0	1	1	UQDECW (scalar) — 32-bit
10	1	0	0	SQINCW (scalar) — 64-bit
10	1	0	1	UQINCW (scalar) — 64-bit
10	1	1	0	SQDECW (scalar) — 64-bit
10	1	1	1	UQDECW (scalar) — 64-bit
11	0	0	0	SQINCD (scalar) — 32-bit
11	0	0	1	UQINCD (scalar) — 32-bit
11	0	1	0	SQDECD (scalar) — 32-bit
11	0	1	1	UQDECD (scalar) — 32-bit
11	1	0	0	SQINCD (scalar) — 64-bit
11	1	0	1	UQINCD (scalar) — 64-bit
11	1	1	0	SQDECD (scalar) — 64-bit
11	1	1	1	UQDECD (scalar) — 64-bit

SVE Bitwise Immediate

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000101								op0		00		op1																			

Decode fields		Instruction details
op0	op1	
11	00	DUPM
!= 11	00	SVE bitwise logical with immediate (unpredicated)
	!= 00	UNALLOCATED

SVE bitwise logical with immediate (unpredicated)

These instructions are under [SVE Bitwise 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
0	0	0	0	0	1	0	1	!= 11	0	0	0	0	imm13												Zdn						
opc																															

The following constraints also apply to this encoding: `opc != 11 && opc != 11`

Decode fields opc	Instruction Details
00	ORR (immediate)
01	EOR (immediate)
10	AND (immediate)

SVE Integer Wide Immediate - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000101										01						op0															

Decode fields op0	Instruction details
0xx	SVE copy integer immediate (predicated)
10x	UNALLOCATED
110	FCPY
111	UNALLOCATED

SVE copy integer immediate (predicated)

These instructions are under [SVE Integer Wide Immediate - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	0	1	Pg			0	M	sh	imm8								Zd						

Decode fields M	Instruction Details
0	CPY (immediate, zeroing)
1	CPY (immediate, merging)

SVE Permute Vector - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000101										1	op0	op1	001110																		

Decode fields op0	Decode fields op1	Instruction details
00	000	DUP (scalar)
00	100	INSR (scalar)
00	x10	UNALLOCATED
00	xx1	UNALLOCATED
01		UNALLOCATED
10	0xx	SVE unpack vector elements
10	100	INSR (SIMD&FP scalar)
10	110	UNALLOCATED

10	1x1	UNALLOCATED
11	000	REV (vector)
11	!= 000	UNALLOCATED

SVE unpack vector elements

These instructions are under [SVE Permute Vector - Unpredicated](#).

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

Decode fields		Instruction Details
U	H	
0	0	SUNPKHI, SUNPKLO — SUNPKLO
0	1	SUNPKHI, SUNPKLO — SUNPKHI
1	0	UUNPKHI, UUNPKLO — UUNPKLO
1	1	UUNPKHI, UUNPKLO — UUNPKHI

SVE Permute Predicate

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000101								op0		1	op1				010		op2						op3								

Decode fields				Instruction details
op0	op1	op2	op3	
00	1000x	0000	0	SVE unpack predicate elements
01	1000x	0000	0	UNALLOCATED
10	1000x	0000	0	UNALLOCATED
11	1000x	0000	0	UNALLOCATED
	0xxxx	xxx0	0	SVE permute predicate elements
	0xxxx	xxx1	0	UNALLOCATED
	10100	0000	0	REV (predicate)
	10101	0000	0	UNALLOCATED
	10x0x	1000	0	UNALLOCATED
	10x0x	x100	0	UNALLOCATED
	10x0x	xx10	0	UNALLOCATED
	10x0x	xxx1	0	UNALLOCATED
	10x1x		0	UNALLOCATED
	11xxx		0	UNALLOCATED
			1	UNALLOCATED

SVE unpack predicate elements

These instructions are under [SVE Permute Predicate](#).

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

Decode fields		Instruction Details
H		
0		PUNPKHI, PUNPKLO — PUNPKLO

Decode fields	Instruction Details
H	
1	PUNPKHI, PUNPKLO — PUNPKHI

SVE permute predicate elements

These instructions are under [SVE Permute Predicate](#).

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

Decode fields	Instruction Details
opc	H
00	0
00	1
01	0
01	1
10	0
10	1
11	

ZIP1, ZIP2 (predicates) — ZIP1
ZIP1, ZIP2 (predicates) — ZIP2
UZP1, UZP2 (predicates) — UZP1
UZP1, UZP2 (predicates) — UZP2
TRN1, TRN2 (predicates) — TRN1
TRN1, TRN2 (predicates) — TRN2
UNALLOCATED

SVE permute vector elements

These instructions are under [SVE encodings](#).

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

Decode fields	Instruction Details
opc	
000	ZIP1, ZIP2 (vectors) — ZIP1
001	ZIP1, ZIP2 (vectors) — ZIP2
010	UZP1, UZP2 (vectors) — UZP1
011	UZP1, UZP2 (vectors) — UZP2
100	TRN1, TRN2 (vectors) — TRN1
101	TRN1, TRN2 (vectors) — TRN2
11x	UNALLOCATED

SVE Permute Vector - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	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	op0	op1	op2	10	op3																

Decode fields				Instruction details
op0	op1	op2	op3	
0	000	0	0	CPY (SIMD&FP scalar)
0	000	1	0	COMPACT
0	000		1	SVE extract element to general register
0	001		0	SVE extract element to SIMD&FP scalar register
0	01x		0	SVE reverse within elements
0	01x		1	UNALLOCATED
0	100	0	1	CPY (scalar)

0	100	1	1	UNALLOCATED
0	100		0	SVE conditionally broadcast element to vector
0	101		0	SVE conditionally extract element to SIMD&FP scalar
0	110	0	0	SPLICE
0	110	0	1	UNALLOCATED
0	110	1		UNALLOCATED
0	111	0	0	UNALLOCATED
0	111	0	1	UNALLOCATED
0	111	1		UNALLOCATED
0	x01		1	UNALLOCATED
1	000		0	UNALLOCATED
1	000		1	SVE conditionally extract element to general register
1	!= 000			UNALLOCATED

SVE extract element to general register

These instructions are under [SVE Permute Vector - Predicated](#).

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

Decode fields	Instruction Details
B	
0	LASTA (scalar)
1	LASTB (scalar)

SVE extract element to SIMD&FP scalar register

These instructions are under [SVE Permute Vector - Predicated](#).

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

Decode fields	Instruction Details
B	
0	LASTA (SIMD&FP scalar)
1	LASTB (SIMD&FP scalar)

SVE reverse within elements

These instructions are under [SVE Permute Vector - Predicated](#).

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

Decode fields	Instruction Details
opc	
00	REVB, REVH, REVW — REVB
01	REVB, REVH, REVW — REVH
10	REVB, REVH, REVW — REVW
11	RBIT

SVE conditionally broadcast element to vector

These instructions are under [SVE Permute Vector - Predicated](#).

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

Decode fields B	Instruction Details
0	CLASTA (vectors)
1	CLASTB (vectors)

SVE conditionally extract element to SIMD&FP scalar

These instructions are under [SVE Permute Vector - Predicated](#).

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

Decode fields B	Instruction Details
0	CLASTA (SIMD&FP scalar)
1	CLASTB (SIMD&FP scalar)

SVE conditionally extract element to general register

These instructions are under [SVE Permute Vector - Predicated](#).

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

Decode fields B	Instruction Details
0	CLASTA (scalar)
1	CLASTB (scalar)

SVE Permute Vector - Extract

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
									000001010	op0	1								000												

Decode fields op0	Instruction details
0	EXT
1	UNALLOCATED

SVE permute vector segments

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	op	1					Zm			0	0	0	opc2				Zn				Zd		

Decode fields op	Decode fields opc2	Instruction Details	Feature
0	000	ZIP1, ZIP2 (vectors) — ZIP1	FEAT_F64MM

Decode fields		Instruction Details	Feature
op	opc2		
0	001	ZIP1, ZIP2 (vectors) — ZIP2	FEAT_F64MM
0	010	UZP1, UZP2 (vectors) — UZP1	FEAT_F64MM
0	011	UZP1, UZP2 (vectors) — UZP2	FEAT_F64MM
0	10x	UNALLOCATED	
0	110	TRN1, TRN2 (vectors) — TRN1	FEAT_F64MM
0	111	TRN1, TRN2 (vectors) — TRN2	FEAT_F64MM
1		UNALLOCATED	

SVE Integer Compare - Vectors

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100100										0							op0														

Decode fields		Instruction details
op0		
0		SVE integer compare vectors
1		SVE integer compare with wide elements

SVE integer compare vectors

These instructions are under [SVE Integer Compare - Vectors](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	0	size	0	Zm			op	0	o2	Pg			Zn			ne	Pd								

Decode fields			Instruction Details
op	o2	ne	
0	0	0	CMP<cc> (vectors) — CMPHS
0	0	1	CMP<cc> (vectors) — CMPHI
0	1	0	CMP<cc> (wide elements) — CMPEQ
0	1	1	CMP<cc> (wide elements) — CMPNE
1	0	0	CMP<cc> (vectors) — CMPGE
1	0	1	CMP<cc> (vectors) — CMPGT
1	1	0	CMP<cc> (vectors) — CMPEQ
1	1	1	CMP<cc> (vectors) — CMPNE

SVE integer compare with wide elements

These instructions are under [SVE Integer Compare - Vectors](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	0	size	0	Zm			U	1	lt	Pg			Zn			ne		Pd							

Decode fields			Instruction Details
U	lt	ne	
0	0	0	CMP<cc> (wide elements) — CMPGE
0	0	1	CMP<cc> (wide elements) — CMPGT
0	1	0	CMP<cc> (wide elements) — CMPLT
0	1	1	CMP<cc> (wide elements) — CMPLE
1	0	0	CMP<cc> (wide elements) — CMPHS

Decode fields			Instruction Details
U	lt	ne	
1	0	1	CMP<cc> (wide elements) — CMPHI
1	1	0	CMP<cc> (wide elements) — CMPLO
1	1	1	CMP<cc> (wide elements) — CMPLS

SVE integer compare with unsigned immediate

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	0	size	1	imm7				lt	Pg		Zn				ne		Pd								

Decode fields		Instruction Details
lt	ne	
0	0	CMP<cc> (immediate) — CMPHS
0	1	CMP<cc> (immediate) — CMPHI
1	0	CMP<cc> (immediate) — CMPLO
1	1	CMP<cc> (immediate) — CMPLS

SVE integer compare with signed immediate

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size	0	imm5			op	0	o2	Pg		Zn				ne		Pd							

Decode fields			Instruction Details
op	o2	ne	
0	0	0	CMP<cc> (immediate) — CMPGE
0	0	1	CMP<cc> (immediate) — CMPGT
0	1	0	CMP<cc> (immediate) — CMPLT
0	1	1	CMP<cc> (immediate) — CMPLT
1	0	0	CMP<cc> (immediate) — CMPEQ
1	0	1	CMP<cc> (immediate) — CMPNE
1	1		UNALLOCATED

SVE predicate logical operations

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	op	S	0	0	Pm			0		1	Pg			o2		Pn			o3		Pd			

Decode fields				Instruction Details
op	S	o2	o3	
0	0	0	0	— not setting the condition flagsAND, ANDS (predicates)
0	0	0	1	— not setting the condition flagsBIC, BICS (predicates)
0	0	1	0	— not setting the condition flagsEOR, EORS (predicates)
0	0	1	1	SEL (predicates)
0	1	0	0	— setting the condition flagsANDSAND, ANDS (predicates)
0	1	0	1	— setting the condition flagsBICSBIC, BICS (predicates)
0	1	1	0	— setting the condition flagsEORSEOR, EORS (predicates)
0	1	1	1	UNALLOCATED

Decode fields				Instruction Details
op	S	o2	o3	
1	0	0	0	— not setting the condition flags ORR, ORRS (predicates)
1	0	0	1	— not setting the condition flags ORN, ORNS (predicates)
1	0	1	0	— not setting the condition flags NOR, NORS
1	0	1	1	— not setting the condition flags NAND, NANDS
1	1	0	0	— setting the condition flags ORRS ORR, ORRS (predicates)
1	1	0	1	— setting the condition flags ORNS ORN, ORNS (predicates)
1	1	1	0	— setting the condition flags NORS NOR, NORS
1	1	1	1	— setting the condition flags NANDS NAND, NANDS

SVE Propagate Break

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100101								00			11			op0																	

Decode fields		Instruction details
op0		
0		SVE propagate break from previous partition
1		UNALLOCATED

SVE propagate break from previous partition

These instructions are under [SVE Propagate Break](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	op	S	0	0	Pm			1	1	Pg			0	Pn			B	Pd						

Decode fields			Instruction Details
op	S	B	
0	0	0	— not setting the condition flags BRKPA, BRKPAS
0	0	1	— not setting the condition flags BRKPB, BRKPBS
0	1	0	— setting the condition flags BRKPAS BRKPA, BRKPAS
0	1	1	— setting the condition flags BRKPBS BRKPB, BRKPBS
1			UNALLOCATED

SVE Partition Break

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100101								op0		01	op1			01	op2							op3									

Decode fields				Instruction details
op0	op1	op2	op3	
0	1000	0	0	SVE propagate break to next partition
0	1000	0	1	UNALLOCATED
0	x000	1		UNALLOCATED
0	x1xx			UNALLOCATED
0	xx1x			UNALLOCATED
0	xxx1			UNALLOCATED

1	0000	1		UNALLOCATED
1	!= 0000			UNALLOCATED
	0000	0		SVE partition break condition

SVE propagate break to next partition

These instructions are under [SVE Partition Break](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	0	S	0	1	1	0	0	0	0	1		Pg		0		Pn		0		Pdm				

Decode fields			Instruction Details			
S						
0			— not setting the condition flags BRKN, BRKNS			
1			— setting the condition flags BRKNSBRKN, BRKNS			

SVE partition break condition

These instructions are under [SVE Partition Break](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	B	S	0	1	0	0	0	0	0	1		Pg		0		Pn		M		Pd				

Decode fields			Instruction Details			
B	S	M				
	1	1	UNALLOCATED			
0	0		— not setting the condition flags BRKA, BRKAS			
0	1	0	— setting the condition flags BRKASBRKA, BRKAS			
1	0		— not setting the condition flags BRKB, BRKBS			
1	1	0	— setting the condition flags BRKBSBRKB, BRKBS			

SVE Predicate Misc

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
											01			op0		11		op1		op2				op3		op4					

Decode fields					Instruction details
op0	op1	op2	op3	op4	
0000		x0		0	SVE predicate test
0100		x0		0	UNALLOCATED
0x10		x0		0	UNALLOCATED
0xx1		x0		0	UNALLOCATED
0xxx		x1		0	UNALLOCATED
1000	000	00		0	SVE predicate first active
1000	000	!= 00		0	UNALLOCATED
1000	100	10	0000	0	SVE predicate zero
1000	100	10	!= 0000	0	UNALLOCATED
1000	110	00		0	SVE predicate read from FFR (predicated)
1001	000	0x		0	UNALLOCATED
1001	000	10		0	PNEXT
1001	000	11		0	UNALLOCATED

1001	100	10		0	UNALLOCATED
1001	110	00	0000	0	SVE predicate read from FFR (unpredicated)
1001	110	00	!= 0000	0	UNALLOCATED
100x	010			0	UNALLOCATED
100x	100	0x		0	SVE predicate initialize
100x	100	11		0	UNALLOCATED
100x	110	!= 00		0	UNALLOCATED
100x	xx1			0	UNALLOCATED
110x				0	UNALLOCATED
1x1x				0	UNALLOCATED
				1	UNALLOCATED

SVE predicate test

These instructions are under [SVE Predicate Misc.](#)

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	op	S	0	1	0	0	0	0	1	1		Pg		0		Pn		0		opc2				

Decode fields			Instruction Details
op	S	opc2	
0	0		UNALLOCATED
0	1	0000	PTEST
0	1	0001	UNALLOCATED
0	1	001x	UNALLOCATED
0	1	01xx	UNALLOCATED
0	1	1xxx	UNALLOCATED
1			UNALLOCATED

SVE predicate first active

These instructions are under [SVE Predicate Misc.](#)

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	0	0	0	0	0	0		Pg		0		Pdn		

Decode fields		Instruction Details
op	S	
0	0	UNALLOCATED
0	1	PFIRST
1		UNALLOCATED

SVE predicate zero

These instructions are under [SVE Predicate Misc.](#)

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0			Pd	

Decode fields		Instruction Details
op	S	
0	0	PFALSE
0	1	UNALLOCATED

Decode fields	Instruction Details
op	S
1	UNALLOCATED

SVE predicate read from FFR (predicated)

These instructions are under [SVE Predicate Misc](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	1	1	0	0	0		Pg		0			Pd		

Decode fields		Instruction Details
op	S	
0	0	not setting the condition flags RDFFR, RDFFRS (predicated)
0	1	setting the condition flags RDFFRSRDFFR, RDFFRS (predicated)
1		UNALLOCATED

SVE predicate read from FFR (unpredicated)

These instructions are under [SVE Predicate Misc](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	op	S	0	1	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0			Pd	

Decode fields		Instruction Details
op	S	
0	0	RDFFR (unpredicated)
0	1	UNALLOCATED
1		UNALLOCATED

SVE predicate initialize

These instructions are under [SVE Predicate Misc](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size		0	1	1	0	0	S	1	1	1	0	0	0		pattern		0				Pd		

Decode fields	Instruction Details
S	
0	not setting the condition flags PTRUE, PTRUES
1	setting the condition flags PTRUES, PTRUE, PTRUES

SVE Integer Compare - Scalars

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	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						00	op0	op1												op2	

Decode fields			Instruction details
op0	op1	op2	
0			SVE integer compare scalar count and limit
1	000	0000	SVE conditionally terminate scalars
1	000	!= 0000	UNALLOCATED
1	!= 000		UNALLOCATED

SVE integer compare scalar count and limit

These instructions are under [SVE Integer Compare - Scalars](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size	1	Rm			0	0	0	sf	U	lt	Rn			eq	Pd								

Decode fields			Instruction Details
U	lt	eq	
	0		UNALLOCATED
0	1	0	WHILELT
0	1	1	WHILELE
1	1	0	WHILELO
1	1	1	WHILELS

SVE conditionally terminate scalars

These instructions are under [SVE Integer Compare - Scalars](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	op	sz	1			Rm			0	0	1	0	0	0				Rn		ne	0	0	0	0

Decode fields		Instruction Details
op	ne	
0		UNALLOCATED
1	0	CTERMEQ, CTERMNE — CTERMEQ
1	1	CTERMEQ, CTERMNE — CTERMNE

SVE Integer Wide Immediate - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	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	op0				op1	11															

Decode fields		Instruction details
op0	op1	
00		SVE integer add/subtract immediate (unpredicated)
01		SVE integer min/max immediate (unpredicated)
10		SVE integer multiply immediate (unpredicated)
11	0	SVE broadcast integer immediate (unpredicated)
11	1	SVE broadcast floating-point immediate (unpredicated)

SVE integer add/subtract immediate (unpredicated)

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size	1	0	0		opc		1	1	sh														Zdn

Decode fields		Instruction Details
opc		
000		ADD (immediate)
001		SUB (immediate)
010		UNALLOCATED
011		SUBR (immediate)

Decode fields opc	Instruction Details
100	SQADD (immediate)
101	UQADD (immediate)
110	SQSUB (immediate)
111	UQSUB (immediate)

SVE integer min/max immediate (unpredicated)

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size	1	0	1	opc		1	1	o2	imm8								Zdn						

Decode fields opc	o2	Instruction Details
0xx	1	UNALLOCATED
000	0	SMAX (immediate)
001	0	UMAX (immediate)
010	0	SMIN (immediate)
011	0	UMIN (immediate)
1xx		UNALLOCATED

SVE integer multiply immediate (unpredicated)

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size		1	1	0	opc		1		1	o2	imm8								Zdn				

Decode fields opc	o2	Instruction Details
000	0	MUL (immediate)
000	1	UNALLOCATED
001		UNALLOCATED
01x		UNALLOCATED
1xx		UNALLOCATED

SVE broadcast integer immediate (unpredicated)

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size		1	1	1	opc		0	1	1	sh	imm8								Zd				

Decode fields opc	Instruction Details
00	DUP (immediate)
01	UNALLOCATED
1x	UNALLOCATED

SVE broadcast floating-point immediate (unpredicated)

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size		1	1	1	opc		1	1	1	o2	imm8								Zd				

Decode fields		Instruction Details
opc	o2	
00	0	FDUP
00	1	UNALLOCATED
01		UNALLOCATED
1x		UNALLOCATED

SVE Predicate Count

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
00100101												100						10						op0											

Decode fields		Instruction details
op0		
0		SVE predicate count
1		UNALLOCATED

SVE predicate count

These instructions are under [SVE Predicate Count](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size		1	0	0	opc		1	0	Pg		0	Pn		Rd									

Decode fields		Instruction Details
opc		
000		CNTP
001		UNALLOCATED
01x		UNALLOCATED
1xx		UNALLOCATED

SVE Inc/Dec by Predicate Count

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
00100101									101		op0		1000		op1																	

Decode fields		Instruction details
op0	op1	
0	0	SVE saturating inc/dec vector by predicate count
0	1	SVE saturating inc/dec register by predicate count
1	0	SVE inc/dec vector by predicate count
1	1	SVE inc/dec register by predicate count

SVE saturating inc/dec vector by predicate count

These instructions are under [SVE Inc/Dec by Predicate Count](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size		1	0	1	0	D	U	1	0	0	0	0	opc		Pm				Zdn				

Decode fields			Instruction Details
D	U	opc	
		01	UNALLOCATED
		1x	UNALLOCATED
0	0	00	SQINCP (vector)
0	1	00	UQINCP (vector)
1	0	00	SQDECP (vector)
1	1	00	UQDECP (vector)

SVE saturating inc/dec register by predicate count

These instructions are under [SVE Inc/Dec by Predicate Count](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size	1	0	1	0	D	U	1	0	0	0	1	sf	op	Pm			Rdn						

Decode fields				Instruction Details
D	U	sf	op	
			1	UNALLOCATED
0	0	0	0	SQINCP (scalar) — 32-bit
0	0	1	0	SQINCP (scalar) — 64-bit
0	1	0	0	UQINCP (scalar) — 32-bit
0	1	1	0	UQINCP (scalar) — 64-bit
1	0	0	0	SQDECP (scalar) — 32-bit
1	0	1	0	SQDECP (scalar) — 64-bit
1	1	0	0	UQDECP (scalar) — 32-bit
1	1	1	0	UQDECP (scalar) — 64-bit

SVE inc/dec vector by predicate count

These instructions are under [SVE Inc/Dec by Predicate Count](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size	1	0	1	1	op	D	1	0	0	0	0	opc2	Pm			Zdn							

Decode fields			Instruction Details
op	D	opc2	
0		01	UNALLOCATED
0		1x	UNALLOCATED
0	0	00	INCP (vector)
0	1	00	DECP (vector)
1			UNALLOCATED

SVE inc/dec register by predicate count

These instructions are under [SVE Inc/Dec by Predicate Count](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	size	1	0	1	1	op	D	1	0	0	0	1	opc2	Pm			Rdn							

Decode fields			Instruction Details
op	D	opc2	
0		01	UNALLOCATED
0		1x	UNALLOCATED

Decode fields			Instruction Details
op	D	opc2	
0	0	00	INCP (scalar)
0	1	00	DECP (scalar)
1			UNALLOCATED

SVE Write FFR

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100101										101		op0	op1			1001				op2				op3						op4	

Decode fields					Instruction details
op0	op1	op2	op3	op4	
0	00	000		00000	SVE FFR write from predicate
1	00	000	0000	00000	SVE FFR initialise
1	00	000	1xxx	00000	UNALLOCATED
1	00	000	x1xx	00000	UNALLOCATED
1	00	000	xx1x	00000	UNALLOCATED
1	00	000	xxx1	00000	UNALLOCATED
	00	000		!= 00000	UNALLOCATED
	00	!= 000			UNALLOCATED
	!= 00				UNALLOCATED

SVE FFR write from predicate

These instructions are under [SVE Write FFR](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	opc	1	0	1	0	0	0	1	0	0	1	0	0	0	0		Pn		0	0	0	0	0	0

Decode fields	Instruction Details
opc	
00	WRFFR
01	UNALLOCATED
1x	UNALLOCATED

SVE FFR initialise

These instructions are under [SVE Write FFR](#).

31	30	29	28	27	26	25	24	23	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	0	0	1	0	1	opc	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Decode fields	Instruction Details
opc	
00	SETFFR
01	UNALLOCATED
1x	UNALLOCATED

SVE Integer Multiply-Add - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01000100										0						0	op0		op1		op2										

Decode fields			Instruction details
op0	op1	op2	
0	000		SVE integer dot product (unpredicated)
0	!= 000		UNALLOCATED
1	0xx		UNALLOCATED
1	10x		UNALLOCATED
1	110		UNALLOCATED
1	111	0	SVE mixed sign dot product
1	111	1	UNALLOCATED

SVE integer dot product (unpredicated)

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	size	0	Zm						0	0	0	0	0	U	Zn						Zda				

Decode fields	Instruction Details
U	
0	SDOT (vectors)
1	UDOT (vectors)

SVE mixed sign dot product

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	size	0	Zm						0	1	1	1	1	0	Zn						Zda					

Decode fields	Instruction Details	Feature
size		
0x	UNALLOCATED	-
10	USDOT (vectors)	FEAT_I8MM
11	UNALLOCATED	-

SVE Multiply - Indexed

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01000100										1							op0		op1												

Decode fields		Instruction details
op0	op1	
000	00	SVE integer dot product (indexed)
000	01	UNALLOCATED
000	10	UNALLOCATED
000	11	SVE mixed sign dot product (indexed)
!= 000		UNALLOCATED

SVE integer dot product (indexed)

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	size	1			opc		0	0	0	0	0	0	U			Zn						Zda		

Decode fields size	U	Instruction Details
0x		UNALLOCATED
10	0	SDOT (indexed) — 32-bit
10	1	UDOT (indexed) — 32-bit
11	0	SDOT (indexed) — 64-bit
11	1	UDOT (indexed) — 64-bit

SVE mixed sign dot product (indexed)

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	size	1			opc		0	0	0	1	1	U			Zn							Zda		

Decode fields size	U	Instruction Details	Feature
0x		UNALLOCATED	⋮
10	0	USDOT (indexed)	FEAT_I8MM
10	1	SUDOT	FEAT_I8MM
11		UNALLOCATED	⋮

SVE Misc

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					01000101					0						10			op0												

Decode fields op0	Instruction details
00xx	UNALLOCATED
010x	UNALLOCATED
0110	SVE integer matrix multiply accumulate
0111	UNALLOCATED
1xxx	UNALLOCATED

SVE integer matrix multiply accumulate

These instructions are under [SVE Misc](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	uns	0			Zm		1	0	0	1	1	0			Zn							Zd		

Decode fields uns	Instruction Details	Feature
00	SMMLA	FEAT_I8MM
01	UNALLOCATED	⋮
10	USMMLA	FEAT_I8MM
11	UMMLA	FEAT_I8MM

SVE floating-point convert precision odd elements

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	opc	0	0	1	0	opc2	1	0	1	Pg				Zn				Zd						

Decode fields opc	Decode fields opc2	Instruction Details	Feature
0x		UNALLOCATED	-
10	0x	UNALLOCATED	-
10	10	BFCVTNT	FEAT_BF16
10	11	UNALLOCATED	-
11		UNALLOCATED	-

SVE floating-point multiply-add (indexed)

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	size	1	opc				0	0	0	0	0	0	op	Zn				Zda						

Decode fields size	Decode fields op	Instruction Details
0x	0	FMLA (indexed) — half-precision
0x	1	FMLS (indexed) — half-precision
10	0	FMLA (indexed) — single-precision
10	1	FMLS (indexed) — single-precision
11	0	FMLA (indexed) — double-precision
11	1	FMLS (indexed) — double-precision

SVE floating-point complex multiply-add (indexed)

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	size	1	opc				0	0	0	1	rot				Zn				Zda					

Decode fields size	Instruction Details
0x	UNALLOCATED
10	FCMLA (indexed) — half-precision
11	FCMLA (indexed) — single-precision

SVE floating-point multiply (indexed)

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	size	1	opc				0	0	1	0	0	0	Zn				Zd							

Decode fields size	Instruction Details
0x	FMUL (indexed) — half-precision
10	FMUL (indexed) — single-precision
11	FMUL (indexed) — double-precision

SVE Floating Point Widening Multiply-Add - Indexed

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100100								op0		1					01	op1	0	op2													

Decode fields			Instruction details
op0	op1	op2	
0	0	00	SVE BFloat16 floating-point dot product (indexed)
0	0	!= 00	UNALLOCATED
0	1		UNALLOCATED
1			SVE floating-point multiply-add long (indexed)

SVE BFloat16 floating-point dot product (indexed)

These instructions are under [SVE Floating Point Widening Multiply-Add - Indexed](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	0	op	1	i2	Zm				0	1	0	0	0	0	Zn				Zda					

Decode fields op	Instruction Details	Feature
0	UNALLOCATED	-
1	BFDOT (indexed)	FEAT_BF16

SVE floating-point multiply-add long (indexed)

These instructions are under [SVE Floating Point Widening Multiply-Add - Indexed](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	1	o2	1	i3h	Zm				0	1	op	0	i3l	T	Zn				Zda					

Decode fields			Instruction Details	Feature
o2	op	T		
0			UNALLOCATED	-
1	0	0	BFMLALB (indexed)	FEAT_BF16
1	0	1	BFMLALT (indexed)	FEAT_BF16
1	1		UNALLOCATED	-

SVE Floating Point Widening Multiply-Add

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100100								op0		1					10	op1	00	op2													

Decode fields			Instruction details
op0	op1	op2	
0	0	0	SVE BFloat16 floating-point dot product
0	0	1	UNALLOCATED
0	1		UNALLOCATED
1			SVE floating-point multiply-add long

SVE BFloat16 floating-point dot product

These instructions are under [SVE Floating Point Widening Multiply-Add](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	0	op	1	Zm					1	0	0	0	0	0	Zn					Zda				

Decode fields op	Instruction Details	Feature
0	UNALLOCATED	-
1	BFDOT (vectors)	FEAT_BF16

SVE floating-point multiply-add long

These instructions are under [SVE Floating Point Widening Multiply-Add](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	1	o2	1	Zm					1	0	op	0	0	T	Zn					Zda				

Decode fields o2 op T	Instruction Details	Feature
0	UNALLOCATED	-
1 0 0	BFMLALB (vectors)	FEAT_BF16
1 0 1	BFMLALT (vectors)	FEAT_BF16
1 1	UNALLOCATED	-

SVE floating point matrix multiply accumulate

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	0	opc	1	Zm						1	1	1	0	0	1	Zn						Zda				

Decode fields opc	Instruction Details	Feature
00	UNALLOCATED	-
01	BFMMLA	FEAT_BF16
10	FMMLA — 32-bit element	FEAT_F32MM
11	FMMLA — 64-bit element	FEAT_F64MM

SVE floating-point compare vectors

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size	0	Zm			op		1	o2	Pg			Zn				o3		Pd					

Decode fields op o2 o3	Instruction Details
0 0 0	FCM<cc> (vectors) — FCMGE
0 0 1	FCM<cc> (vectors) — FCMGT
0 1 0	FCM<cc> (vectors) — FCMEQ
0 1 1	FCM<cc> (vectors) — FCMNE
1 0 0	FCM<cc> (vectors) — FCMUO
1 0 1	FAC<cc> — FACGE
1 1 0	UNALLOCATED

Decode fields			Instruction Details
op	o2	o3	
1	1	1	FAC<cc> — FACGT

SVE floating-point arithmetic (unpredicated)

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size	0			Zm				0	0	0	opc				Zn						Zd		

Decode fields		Instruction Details
opc		
000		FADD (vectors, unpredicated)
001		FSUB (vectors, unpredicated)
010		FMUL (vectors, unpredicated)
011		FTSMUL
10x		UNALLOCATED
110		FRECPS
111		FRSQRTS

SVE Floating Point Arithmetic - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	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	0	1		0	op0					100			op1			op2									

Decode fields			Instruction details
op0	op1	op2	
0x			SVE floating-point arithmetic (predicated)
10	000		FTMAD
10	!= 000		UNALLOCATED
11		0000	SVE floating-point arithmetic with immediate (predicated)
11		!= 0000	UNALLOCATED

SVE floating-point arithmetic (predicated)

These instructions are under [SVE Floating Point Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size	0	0		opc				1	0	0	Pg				Zm						Zdn		

Decode fields		Instruction Details
opc		
0000		FADD (vectors, predicated)
0001		FSUB (vectors, predicated)
0010		FMUL (vectors, predicated)
0011		FSUBR (vectors)
0100		FMAXNM (vectors)
0101		FMINNM (vectors)
0110		FMAX (vectors)
0111		FMIN (vectors)
1000		FABD

Decode fields opc	Instruction Details
1001	FSCALE
1010	FMULX
1011	UNALLOCATED
1100	FDIVR
1101	FDIV
111x	UNALLOCATED

SVE floating-point arithmetic with immediate (predicated)

These instructions are under [SVE Floating Point Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size	0	1	1	opc	1	0	0	Pg	0	0	0	0	il	Zdn									

Decode fields opc	Instruction Details
000	FADD (immediate)
001	FSUB (immediate)
010	FMUL (immediate)
011	FSUBR (immediate)
100	FMAXNM (immediate)
101	FMINNM (immediate)
110	FMAX (immediate)
111	FMIN (immediate)

SVE Floating Point Unary Operations - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100101									0	op0				101																	

Decode fields op0	Instruction details
00x	SVE floating-point round to integral value
010	SVE floating-point convert precision
011	SVE floating-point unary operations
10x	SVE integer convert to floating-point
11x	SVE floating-point convert to integer

SVE floating-point round to integral value

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size	0	0	0	opc	1	0	1	Pg	Zn				Zd										

Decode fields opc	Instruction Details
000	FRINT<r> — nearest with ties to even
001	FRINT<r> — toward plus infinity
010	FRINT<r> — toward minus infinity

Decode fields opc	Instruction Details
011	FRINT<r> — toward zero
100	FRINT<r> — nearest with ties to away
101	UNALLOCATED
110	FRINT<r> — current mode signalling inexact
111	FRINT<r> — current mode

SVE floating-point convert precision

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1		opc	0	0	1	0		opc2	1	0	1		Pg				Zn					Zd		

Decode fields opc	Decode fields opc2	Instruction Details	Feature
0x		UNALLOCATED	—
10	00	FCVT — single-precision to half-precision	—
10	01	FCVT — half-precision to single-precision	—
10	10	BFCVT	FEAT_BF16
10	11	UNALLOCATED	—
11	00	FCVT — double-precision to half-precision	—
11	01	FCVT — half-precision to double-precision	—
11	10	FCVT — double-precision to single-precision	—
11	11	FCVT — single-precision to double-precision	—

SVE floating-point unary operations

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1		size	0	0	1	1		opc	1	0	1		Pg				Zn					Zd		

Decode fields opc	Instruction Details
00	FRECPX
01	FSQRT
1x	UNALLOCATED

SVE integer convert to floating-point

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1		opc	0	1	0		opc2	U	1	0	1		Pg				Zn					Zd		

Decode fields opc	Decode fields opc2	Decode fields U	Instruction Details
00			UNALLOCATED
01	00		UNALLOCATED
01	01	0	SCVTF — 16-bit to half-precision
01	01	1	UCVTF — 16-bit to half-precision
01	10	0	SCVTF — 32-bit to half-precision

Decode fields			Instruction Details
opc	opc2	U	
01	10	1	UCVTF — 32-bit to half-precision
01	11	0	SCVTF — 64-bit to half-precision
01	11	1	UCVTF — 64-bit to half-precision
10	0x		UNALLOCATED
10	10	0	SCVTF — 32-bit to single-precision
10	10	1	UCVTF — 32-bit to single-precision
10	11		UNALLOCATED
11	00	0	SCVTF — 32-bit to double-precision
11	00	1	UCVTF — 32-bit to double-precision
11	01		UNALLOCATED
11	10	0	SCVTF — 64-bit to single-precision
11	10	1	UCVTF — 64-bit to single-precision
11	11	0	SCVTF — 64-bit to double-precision
11	11	1	UCVTF — 64-bit to double-precision

SVE floating-point convert to integer

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	opc	0	1	1	opc2	U	1	0	1	Pg			Zn					Zd						

Decode fields			Instruction Details
opc	opc2	U	
00			UNALLOCATED
01	00		UNALLOCATED
01	01	0	FCVTZS — half-precision to 16-bit
01	01	1	FCVTZU — half-precision to 16-bit
01	10	0	FCVTZS — half-precision to 32-bit
01	10	1	FCVTZU — half-precision to 32-bit
01	11	0	FCVTZS — half-precision to 64-bit
01	11	1	FCVTZU — half-precision to 64-bit
10	0x		UNALLOCATED
10	10	0	FCVTZS — single-precision to 32-bit
10	10	1	FCVTZU — single-precision to 32-bit
10	11		UNALLOCATED
11	00	0	FCVTZS — double-precision to 32-bit
11	00	1	FCVTZU — double-precision to 32-bit
11	01		UNALLOCATED
11	10	0	FCVTZS — single-precision to 64-bit
11	10	1	FCVTZU — single-precision to 64-bit
11	11	0	FCVTZS — double-precision to 64-bit
11	11	1	FCVTZU — double-precision to 64-bit

SVE floating-point recursive reduction

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size	0	0	0	opc	0	0	1	Pg			Zn			Vd									

Decode fields opc	Instruction Details
000	FADDV
001	UNALLOCATED
01x	UNALLOCATED
100	FMAXNMV
101	FMINNMV
110	FMAXV
111	FMINV

SVE Floating Point Unary Operations - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100101												001						0011		op0											

Decode fields op0	Instruction details
00	SVE floating-point reciprocal estimate (unpredicated)
!= 00	UNALLOCATED

SVE floating-point reciprocal estimate (unpredicated)

These instructions are under [SVE Floating Point Unary Operations - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size		0	0	1	opc		0		0	1	1	0	0	Zn				Zd					

Decode fields opc	Instruction Details
0xx	UNALLOCATED
10x	UNALLOCATED
110	FRECPE
111	FRSQRT

SVE Floating Point Compare - with Zero

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100101												010		op0						001											

Decode fields op0	Instruction details
0	SVE floating-point compare with zero
1	UNALLOCATED

SVE floating-point compare with zero

These instructions are under [SVE Floating Point Compare - with Zero](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size		0	1	0	0	eq		lt		0		0	1	Pg		Zn				ne		Pd	

Decode fields			Instruction Details
eq	lt	ne	
0	0	0	FCM<cc> (zero) — FCMGE
0	0	1	FCM<cc> (zero) — FCMGT
0	1	0	FCM<cc> (zero) — FCMLT
0	1	1	FCM<cc> (zero) — FCMLE
1		1	UNALLOCATED
1	0	0	FCM<cc> (zero) — FCMEQ
1	1	0	FCM<cc> (zero) — FCMNE

SVE Floating Point Accumulating Reduction

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100101										011	op0			001																	

Decode fields		Instruction details
op0		
0		SVE floating-point serial reduction (predicated)
1		UNALLOCATED

SVE floating-point serial reduction (predicated)

These instructions are under [SVE Floating Point Accumulating Reduction](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size		0	1	1	0	opc		0	0	1		Pg					Zm				Vdn		

Decode fields	Instruction Details
opc	
00	FADDA
01	UNALLOCATED
1x	UNALLOCATED

SVE Floating Point Multiply-Add

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100101										1				op0																	

Decode fields		Instruction details
op0		
0		SVE floating-point multiply-accumulate writing addend
1		SVE floating-point multiply-accumulate writing multiplicand

SVE floating-point multiply-accumulate writing addend

These instructions are under [SVE Floating Point Multiply-Add](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size	1	Zm			0	opc	Pg		Zn			Zda											

Decode fields opc	Instruction Details
00	FMLA (vectors)
01	FMLS (vectors)
10	FNMLA
11	FNMLS

SVE floating-point multiply-accumulate writing multiplicand

These instructions are under [SVE Floating Point Multiply-Add](#).

31	30	29	28	27	26	25	24	23	22	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	0	1	0	1	size	1	Za			1	opc	Pg			Zm					Zdn								

Decode fields opc	Instruction Details
00	FMAD
01	FMSB
10	FNMAAD
11	FNMSB

SVE Memory - 32-bit Gather and Unsized Contiguous

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1000010							op0		op1							op2													op3						

Decode fields				Instruction details	
op0	op1	op2	op3		
00	x1	0xx	0	SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)	
00	x1	0xx	1	UNALLOCATED	
01	x1	0xx		SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)	
10	x1	0xx		SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)	
11	0x	000	0	LDR (predicate)	
11	0x	000	1	UNALLOCATED	
11	0x	010		LDR (vector)	
11	0x	0x1		UNALLOCATED	
11	1x	0xx	0	SVE contiguous prefetch (scalar plus immediate)	
11	1x	0xx	1	UNALLOCATED	
!= 11	x0	0xx		SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)	
	00	10x		UNALLOCATED	
	00	110	0	SVE contiguous prefetch (scalar plus scalar)	
	00	111	0	SVE 32-bit gather prefetch (vector plus immediate)	
	00	11x	1	UNALLOCATED	
	01	1xx		SVE 32-bit gather load (vector plus immediate)	
	1x	1xx		SVE load and broadcast element	

SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	0	0	xs	1			Zm		0	msz		Pg						Rn		0			prfop		

Decode fields msz	Instruction Details
00	PRFB (scalar plus vector)
01	PRFH (scalar plus vector)
10	PRFW (scalar plus vector)
11	PRFD (scalar plus vector)

SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	0	1	xs	1			Zm		0	U	ff		Pg						Rn				Zt		

Decode fields U	ff	Instruction Details
0	0	LD1SH (scalar plus vector)
0	1	LDF1SH (scalar plus vector)
1	0	LD1H (scalar plus vector)
1	1	LDF1H (scalar plus vector)

SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	1	0	xs	1			Zm		0	U	ff		Pg						Rn				Zt		

Decode fields U	ff	Instruction Details
0		UNALLOCATED
1	0	LD1W (scalar plus vector)
1	1	LDF1W (scalar plus vector)

SVE contiguous prefetch (scalar plus immediate)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	1	1	1				imm6		0	msz		Pg							Rn		0			prfop	

Decode fields msz	Instruction Details
00	PRFB (scalar plus immediate)
01	PRFH (scalar plus immediate)
10	PRFW (scalar plus immediate)
11	PRFD (scalar plus immediate)

SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	!= 11	xs	0	Zm						0	U	ff	Pg			Rn						Zt			

opc

The following constraints also apply to this encoding: opc != 11 && opc != 11

Decode fields			Instruction Details
opc	U	ff	
00	0	0	LD1SB (scalar plus vector)
00	0	1	LDFF1SB (scalar plus vector)
00	1	0	LD1B (scalar plus vector)
00	1	1	LDFF1B (scalar plus vector)
01	0	0	LD1SH (scalar plus vector)
01	0	1	LDFF1SH (scalar plus vector)
01	1	0	LD1H (scalar plus vector)
01	1	1	LDFF1H (scalar plus vector)
10	0		UNALLOCATED
10	1	0	LD1W (scalar plus vector)
10	1	1	LDFF1W (scalar plus vector)

SVE contiguous prefetch (scalar plus scalar)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	0	0	Rm						1	1	0	Pg			Rn						0	prfop		

Decode fields		Instruction Details
msz		
00		PRFB (scalar plus scalar)
01		PRFH (scalar plus scalar)
10		PRFW (scalar plus scalar)
11		PRFD (scalar plus scalar)

SVE 32-bit gather prefetch (vector plus immediate)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	0	0	imm5						1	1	1	Pg			Zn						0	prfop		

Decode fields		Instruction Details
msz		
00		PRFB (vector plus immediate)
01		PRFH (vector plus immediate)
10		PRFW (vector plus immediate)
11		PRFD (vector plus immediate)

SVE 32-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	0	1	imm5						1	U	ff	Pg			Zn						Zt			

Decode fields			Instruction Details
msz	U	ff	
00	0	0	LD1SB (vector plus immediate)
00	0	1	LDFF1SB (vector plus immediate)
00	1	0	LD1B (vector plus immediate)
00	1	1	LDFF1B (vector plus immediate)
01	0	0	LD1SH (vector plus immediate)
01	0	1	LDFF1SH (vector plus immediate)
01	1	0	LD1H (vector plus immediate)
01	1	1	LDFF1H (vector plus immediate)
10	0		UNALLOCATED
10	1	0	LD1W (vector plus immediate)
10	1	1	LDFF1W (vector plus immediate)
11			UNALLOCATED

SVE load and broadcast element

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	dtypeh	1	imm6						1	dtypel	Pg			Rn				Zt							

Decode fields		Instruction Details
dtypeh	dtypel	
00	00	LD1RB — 8-bit element
00	01	LD1RB — 16-bit element
00	10	LD1RB — 32-bit element
00	11	LD1RB — 64-bit element
01	00	LD1RSW
01	01	LD1RH — 16-bit element
01	10	LD1RH — 32-bit element
01	11	LD1RH — 64-bit element
10	00	LD1RSH — 64-bit element
10	01	LD1RSH — 32-bit element
10	10	LD1RW — 32-bit element
10	11	LD1RW — 64-bit element
11	00	LD1RSB — 64-bit element
11	01	LD1RSB — 32-bit element
11	10	LD1RSB — 16-bit element
11	11	LD1RD

SVE Memory - Contiguous Load

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1010010								op0		op1							op2														

Decode fields			Instruction details
op0	op1	op2	
00	0	111	SVE contiguous non-temporal load (scalar plus immediate)
00		110	SVE contiguous non-temporal load (scalar plus scalar)

!= 00	0	111	SVE load multiple structures (scalar plus immediate)
!= 00		110	SVE load multiple structures (scalar plus scalar)
	0	001	SVE load and broadcast quadword (scalar plus immediate)
	0	101	SVE contiguous load (scalar plus immediate)
	1	001	UNALLOCATED
	1	101	SVE contiguous non-fault load (scalar plus immediate)
	1	111	UNALLOCATED
		000	SVE load and broadcast quadword (scalar plus scalar)
		010	SVE contiguous load (scalar plus scalar)
		011	SVE contiguous first-fault load (scalar plus scalar)
		100	UNALLOCATED

SVE contiguous non-temporal load (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	0	0	0		imm4				1	1	1		Pg				Rn					Zt		

Decode fields msz	Instruction Details
00	LDNT1B (scalar plus immediate)
01	LDNT1H (scalar plus immediate)
10	LDNT1W (scalar plus immediate)
11	LDNT1D (scalar plus immediate)

SVE contiguous non-temporal load (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	0	0			Rm				1	1	0		Pg				Rn					Zt		

Decode fields msz	Instruction Details
00	LDNT1B (scalar plus scalar)
01	LDNT1H (scalar plus scalar)
10	LDNT1W (scalar plus scalar)
11	LDNT1D (scalar plus scalar)

SVE load multiple structures (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	!= 00	0		imm4					1	1	1		Pg				Rn					Zt		
opc																															

The following constraints also apply to this encoding: `opc != 00 && opc != 00`

Decode fields msz opc	Instruction Details
00 01	LD2B (scalar plus immediate)
00 10	LD3B (scalar plus immediate)

Decode fields		Instruction Details
msz	opc	
00	11	LD4B (scalar plus immediate)
01	01	LD2H (scalar plus immediate)
01	10	LD3H (scalar plus immediate)
01	11	LD4H (scalar plus immediate)
10	01	LD2W (scalar plus immediate)
10	10	LD3W (scalar plus immediate)
10	11	LD4W (scalar plus immediate)
11	01	LD2D (scalar plus immediate)
11	10	LD3D (scalar plus immediate)
11	11	LD4D (scalar plus immediate)

SVE load multiple structures (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz		!= 00		Rm			1	1	0	Pg		Rn			Zt									
opc																															

The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields		Instruction Details
msz	opc	
00	01	LD2B (scalar plus scalar)
00	10	LD3B (scalar plus scalar)
00	11	LD4B (scalar plus scalar)
01	01	LD2H (scalar plus scalar)
01	10	LD3H (scalar plus scalar)
01	11	LD4H (scalar plus scalar)
10	01	LD2W (scalar plus scalar)
10	10	LD3W (scalar plus scalar)
10	11	LD4W (scalar plus scalar)
11	01	LD2D (scalar plus scalar)
11	10	LD3D (scalar plus scalar)
11	11	LD4D (scalar plus scalar)

SVE load and broadcast quadword (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz		ssz		0	imm4			0		0	1	Pg			Rn			Zt						

Decode fields		Instruction Details	Feature
msz	ssz		
	1x	UNALLOCATED	█
00	00	LD1RQB (scalar plus immediate)	█
00	01	LD1ROB (scalar plus immediate)	FEAT_F64MM
01	00	LD1RQH (scalar plus immediate)	█
01	01	LD1ROH (scalar plus immediate)	FEAT_F64MM
10	00	LD1RQW (scalar plus immediate)	█

Decode fields		Instruction Details	Feature
msz	ssz		
10	01	LD1ROW (scalar plus immediate)	FEAT_F64MM
11	00	LD1RQD (scalar plus immediate)	
11	01	LD1ROD (scalar plus immediate)	FEAT_F64MM

SVE contiguous load (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	dtype			0	imm4			1	0	1	Pg			Rn			Zt								

Decode fields	Instruction Details
dtype	
0000	LD1B (scalar plus immediate) — 8-bit element
0001	LD1B (scalar plus immediate) — 16-bit element
0010	LD1B (scalar plus immediate) — 32-bit element
0011	LD1B (scalar plus immediate) — 64-bit element
0100	LD1SW (scalar plus immediate)
0101	LD1H (scalar plus immediate) — 16-bit element
0110	LD1H (scalar plus immediate) — 32-bit element
0111	LD1H (scalar plus immediate) — 64-bit element
1000	LD1SH (scalar plus immediate) — 64-bit element
1001	LD1SH (scalar plus immediate) — 32-bit element
1010	LD1W (scalar plus immediate) — 32-bit element
1011	LD1W (scalar plus immediate) — 64-bit element
1100	LD1SB (scalar plus immediate) — 64-bit element
1101	LD1SB (scalar plus immediate) — 32-bit element
1110	LD1SB (scalar plus immediate) — 16-bit element
1111	LD1D (scalar plus immediate)

SVE contiguous non-fault load (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	dtype			1	imm4			1	0	1	Pg			Rn			Zt								

Decode fields	Instruction Details
dtype	
0000	LDNF1B — 8-bit element
0001	LDNF1B — 16-bit element
0010	LDNF1B — 32-bit element
0011	LDNF1B — 64-bit element
0100	LDNF1SW
0101	LDNF1H — 16-bit element
0110	LDNF1H — 32-bit element
0111	LDNF1H — 64-bit element
1000	LDNF1SH — 64-bit element
1001	LDNF1SH — 32-bit element
1010	LDNF1W — 32-bit element

Decode fields dtype	Instruction Details
1011	LDNF1W — 64-bit element
1100	LDNF1SB — 64-bit element
1101	LDNF1SB — 32-bit element
1110	LDNF1SB — 16-bit element
1111	LDNF1D

SVE load and broadcast quadword (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz		ssz		Rm				0 0 0			Pg		Rn				Zt							

Decode fields msz ssz		Instruction Details	Feature
	1x	UNALLOCATED	—
00	00	LD1RQB (scalar plus scalar)	—
00	01	LD1ROB (scalar plus scalar)	FEAT_F64MM
01	00	LD1RQH (scalar plus scalar)	—
01	01	LD1ROH (scalar plus scalar)	FEAT_F64MM
10	00	LD1RQW (scalar plus scalar)	—
10	01	LD1ROW (scalar plus scalar)	FEAT_F64MM
11	00	LD1RQD (scalar plus scalar)	—
11	01	LD1ROD (scalar plus scalar)	FEAT_F64MM

SVE contiguous load (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	dtype				Rm				0	1	0	Pg		Rn				Zt							

Decode fields dtype	Instruction Details
0000	LD1B (scalar plus scalar) — 8-bit element
0001	LD1B (scalar plus scalar) — 16-bit element
0010	LD1B (scalar plus scalar) — 32-bit element
0011	LD1B (scalar plus scalar) — 64-bit element
0100	LD1SW (scalar plus scalar)
0101	LD1H (scalar plus scalar) — 16-bit element
0110	LD1H (scalar plus scalar) — 32-bit element
0111	LD1H (scalar plus scalar) — 64-bit element
1000	LD1SH (scalar plus scalar) — 64-bit element
1001	LD1SH (scalar plus scalar) — 32-bit element
1010	LD1W (scalar plus scalar) — 32-bit element
1011	LD1W (scalar plus scalar) — 64-bit element
1100	LD1SB (scalar plus scalar) — 64-bit element
1101	LD1SB (scalar plus scalar) — 32-bit element
1110	LD1SB (scalar plus scalar) — 16-bit element
1111	LD1D (scalar plus scalar)

SVE contiguous first-fault load (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	dtype				Rm				0	1	1	Pg			Rn				Zt						

Decode fields dtype	Instruction Details
0000	LDF1B (scalar plus scalar) — 8-bit element
0001	LDF1B (scalar plus scalar) — 16-bit element
0010	LDF1B (scalar plus scalar) — 32-bit element
0011	LDF1B (scalar plus scalar) — 64-bit element
0100	LDF1SW (scalar plus scalar)
0101	LDF1H (scalar plus scalar) — 16-bit element
0110	LDF1H (scalar plus scalar) — 32-bit element
0111	LDF1H (scalar plus scalar) — 64-bit element
1000	LDF1SH (scalar plus scalar) — 64-bit element
1001	LDF1SH (scalar plus scalar) — 32-bit element
1010	LDF1W (scalar plus scalar) — 32-bit element
1011	LDF1W (scalar plus scalar) — 64-bit element
1100	LDF1SB (scalar plus scalar) — 64-bit element
1101	LDF1SB (scalar plus scalar) — 32-bit element
1110	LDF1SB (scalar plus scalar) — 16-bit element
1111	LDF1D (scalar plus scalar)

SVE Memory - 64-bit Gather

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1100010							op0		op1						op2									op3							

Decode fields				Instruction details
op0	op1	op2	op3	
00	01	0xx	1	UNALLOCATED
00	11	1xx	0	SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)
00	11		1	UNALLOCATED
00	x1	0xx	0	SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)
!= 00	11	1xx		SVE 64-bit gather load (scalar plus 64-bit scaled offsets)
!= 00	x1	0xx		SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)
	00	10x		UNALLOCATED
	00	110		UNALLOCATED
	00	111	0	SVE 64-bit gather prefetch (vector plus immediate)
	00	111	1	UNALLOCATED
	01	1xx		SVE 64-bit gather load (vector plus immediate)
	10	1xx		SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)
	x0	0xx		SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)

SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	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	0	1	0	0	0	1	1	Zm				1	msz		Pg		Rn				0	prfop						

Decode fields msz	Instruction Details
00	PRFB (scalar plus vector)
01	PRFH (scalar plus vector)
10	PRFW (scalar plus vector)
11	PRFD (scalar plus vector)

SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	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	0	1	0	0	0	xs	1	Zm				0	msz		Pg		Rn				0	prfop						

Decode fields msz	Instruction Details
00	PRFB (scalar plus vector)
01	PRFH (scalar plus vector)
10	PRFW (scalar plus vector)
11	PRFD (scalar plus vector)

SVE 64-bit gather load (scalar plus 64-bit scaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	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	0	1	0	!= 00		1	1	Zm				1	U	ff	Pg		Rn				Zt							
opc																															

The following constraints also apply to this encoding: `opc != 00 && opc != 00`

Decode fields opc	U	ff	Instruction Details
01	0	0	LD1SH (scalar plus vector)
01	0	1	LDFF1SH (scalar plus vector)
01	1	0	LD1H (scalar plus vector)
01	1	1	LDFF1H (scalar plus vector)
10	0	0	LD1SW (scalar plus vector)
10	0	1	LDFF1SW (scalar plus vector)
10	1	0	LD1W (scalar plus vector)
10	1	1	LDFF1W (scalar plus vector)
11	0		UNALLOCATED
11	1	0	LD1D (scalar plus vector)
11	1	1	LDFF1D (scalar plus vector)

SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	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	0	1	0	!= 00		xs	1	Zm				0	U	ff	Pg		Rn				Zt							
opc																															

The following constraints also apply to this encoding: `opc != 00 && opc != 00`

Decode fields			Instruction Details
opc	U	ff	
01	0	0	LD1SH (scalar plus vector)
01	0	1	LDFF1SH (scalar plus vector)
01	1	0	LD1H (scalar plus vector)
01	1	1	LDFF1H (scalar plus vector)
10	0	0	LD1SW (scalar plus vector)
10	0	1	LDFF1SW (scalar plus vector)
10	1	0	LD1W (scalar plus vector)
10	1	1	LDFF1W (scalar plus vector)
11	0		UNALLOCATED
11	1	0	LD1D (scalar plus vector)
11	1	1	LDFF1D (scalar plus vector)

SVE 64-bit gather prefetch (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	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	0	1	0	msz	0	0	imm5					1	1	1	Pg		Zn				0	prfop						

Decode fields		Instruction Details
msz		
00		PRFB (vector plus immediate)
01		PRFH (vector plus immediate)
10		PRFW (vector plus immediate)
11		PRFD (vector plus immediate)

SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	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	0	1	0	msz		0	1	imm5					1	U	ff	Pg		Zn				Zt						

Decode fields			Instruction Details
msz	U	ff	
00	0	0	LD1SB (vector plus immediate)
00	0	1	LDFF1SB (vector plus immediate)
00	1	0	LD1B (vector plus immediate)
00	1	1	LDFF1B (vector plus immediate)
01	0	0	LD1SH (vector plus immediate)
01	0	1	LDFF1SH (vector plus immediate)
01	1	0	LD1H (vector plus immediate)
01	1	1	LDFF1H (vector plus immediate)
10	0	0	LD1SW (vector plus immediate)
10	0	1	LDFF1SW (vector plus immediate)
10	1	0	LD1W (vector plus immediate)
10	1	1	LDFF1W (vector plus immediate)
11	0		UNALLOCATED
11	1	0	LD1D (vector plus immediate)

Decode fields			Instruction Details
msz	U	ff	
11	1	1	LDF1D (vector plus immediate)

SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	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	0	1	0	msz		1	0	Zm				1	U	ff	Pg			Rn				Zt						

Decode fields			Instruction Details
msz	U	ff	
00	0	0	LD1SB (scalar plus vector)
00	0	1	LDF1SB (scalar plus vector)
00	1	0	LD1B (scalar plus vector)
00	1	1	LDF1B (scalar plus vector)
01	0	0	LD1SH (scalar plus vector)
01	0	1	LDF1SH (scalar plus vector)
01	1	0	LD1H (scalar plus vector)
01	1	1	LDF1H (scalar plus vector)
10	0	0	LD1SW (scalar plus vector)
10	0	1	LDF1SW (scalar plus vector)
10	1	0	LD1W (scalar plus vector)
10	1	1	LDF1W (scalar plus vector)
11	0		UNALLOCATED
11	1	0	LD1D (scalar plus vector)
11	1	1	LDF1D (scalar plus vector)

SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	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	0	1	0	msz		xs	0	Zm				0	U	ff	Pg			Rn				Zt						

Decode fields			Instruction Details
msz	U	ff	
00	0	0	LD1SB (scalar plus vector)
00	0	1	LDF1SB (scalar plus vector)
00	1	0	LD1B (scalar plus vector)
00	1	1	LDF1B (scalar plus vector)
01	0	0	LD1SH (scalar plus vector)
01	0	1	LDF1SH (scalar plus vector)
01	1	0	LD1H (scalar plus vector)
01	1	1	LDF1H (scalar plus vector)
10	0	0	LD1SW (scalar plus vector)
10	0	1	LDF1SW (scalar plus vector)
10	1	0	LD1W (scalar plus vector)
10	1	1	LDF1W (scalar plus vector)
11	0		UNALLOCATED
11	1	0	LD1D (scalar plus vector)

Decode fields			Instruction Details
msz	U	ff	
11	1	1	LDFFD1D (scalar plus vector)

SVE Memory - Contiguous Store and Unsized Contiguous

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010								op0								0	op1	0									op2				

Decode fields			Instruction details
op0	op1	op2	
0xx	0		UNALLOCATED
10x	0		UNALLOCATED
110	0	0	STR (predicate)
110	0	1	UNALLOCATED
110	1		STR (vector)
111	0		UNALLOCATED
!= 110	1		SVE contiguous store (scalar plus scalar)

SVE contiguous store (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Store and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	!= 110	o2	Rm				0	1	0	Pg		Rn				Zt									
opc																															

The following constraints also apply to this encoding: `opc != 110 && opc != 110`

Decode fields		Instruction Details
opc	o2	
00x		ST1B (scalar plus scalar)
01x		ST1H (scalar plus scalar)
10x		ST1W (scalar plus scalar)
111	0	UNALLOCATED
111	1	ST1D (scalar plus scalar)

SVE Memory - Non-temporal and Multi-register Store

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010								op0								0	op1	1													

Decode fields		Instruction details
op0	op1	
00	1	SVE contiguous non-temporal store (scalar plus scalar)
!= 00	1	SVE store multiple structures (scalar plus scalar)
	0	UNALLOCATED

SVE contiguous non-temporal store (scalar plus scalar)

These instructions are under [SVE Memory - Non-temporal and Multi-register Store](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz		0	0	Rm				0	1	1	Pg		Rn				Zt							

Decode fields msz	Instruction Details
00	STNT1B (scalar plus scalar)
01	STNT1H (scalar plus scalar)
10	STNT1W (scalar plus scalar)
11	STNT1D (scalar plus scalar)

SVE store multiple structures (scalar plus scalar)

These instructions are under [SVE Memory - Non-temporal and Multi-register Store](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz		!= 00		Rm				0		1		1		Pg		Rn				Zt				
opc																															

The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields msz	opc	Instruction Details
00	01	ST2B (scalar plus scalar)
00	10	ST3B (scalar plus scalar)
00	11	ST4B (scalar plus scalar)
01	01	ST2H (scalar plus scalar)
01	10	ST3H (scalar plus scalar)
01	11	ST4H (scalar plus scalar)
10	01	ST2W (scalar plus scalar)
10	10	ST3W (scalar plus scalar)
10	11	ST4W (scalar plus scalar)
11	01	ST2D (scalar plus scalar)
11	10	ST3D (scalar plus scalar)
11	11	ST4D (scalar plus scalar)

SVE Memory - Scatter with Optional Sign Extend

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010								op0								1			0												

Decode fields op0	Instruction details
00	SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)
01	SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)
10	SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)
11	SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under [SVE Memory - Scatter with Optional Sign Extend](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	0	0		Zm		1	xs	0		Pg			Rn								Zt			

Decode fields msz	Instruction Details
00	ST1B (scalar plus vector)
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	ST1D (scalar plus vector)

SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)

These instructions are under [SVE Memory - Scatter with Optional Sign Extend](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	0	1		Zm		1	xs	0		Pg			Rn								Zt			

Decode fields msz	Instruction Details
00	UNALLOCATED
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	ST1D (scalar plus vector)

SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)

These instructions are under [SVE Memory - Scatter with Optional Sign Extend](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	1	0		Zm		1	xs	0		Pg			Rn								Zt			

Decode fields msz	Instruction Details
00	ST1B (scalar plus vector)
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	UNALLOCATED

SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

These instructions are under [SVE Memory - Scatter with Optional Sign Extend](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	1	1		Zm		1	xs	0		Pg			Rn								Zt			

Decode fields msz	Instruction Details
00	UNALLOCATED
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	UNALLOCATED

SVE Memory - Scatter

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010									op0							101															

Decode fields op0	Instruction details
00	SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)
01	SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)
10	SVE 64-bit scatter store (vector plus immediate)
11	SVE 32-bit scatter store (vector plus immediate)

SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)

These instructions are under [SVE Memory - Scatter](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz	0	0	Zm						1	0	1	Pg			Rn				Zt					

Decode fields msz	Instruction Details
00	ST1B (scalar plus vector)
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	ST1D (scalar plus vector)

SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)

These instructions are under [SVE Memory - Scatter](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz		0	1	Zm					1	0	1	Pg			Rn				Zt					

Decode fields msz	Instruction Details
00	UNALLOCATED
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	ST1D (scalar plus vector)

SVE 64-bit scatter store (vector plus immediate)

These instructions are under [SVE Memory - Scatter](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz		1	0	imm5					1	0	1	Pg			Zn				Zt					

Decode fields msz	Instruction Details
00	ST1B (vector plus immediate)
01	ST1H (vector plus immediate)
10	ST1W (vector plus immediate)
11	ST1D (vector plus immediate)

SVE 32-bit scatter store (vector plus immediate)

These instructions are under [SVE Memory - Scatter](#).

31	30	29	28	27	26	25	24	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	0	0	1	0	msz		1	1	imm5				1	0	1	Pg			Zn				Zt						

Decode fields msz	Instruction Details
00	ST1B (vector plus immediate)
01	ST1H (vector plus immediate)
10	ST1W (vector plus immediate)
11	UNALLOCATED

SVE Memory - Contiguous Store with Immediate Offset

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010								op0	op1					111																	

Decode fields op0	op1	Instruction details
00	1	SVE contiguous non-temporal store (scalar plus immediate)
!= 00	1	SVE store multiple structures (scalar plus immediate)
	0	SVE contiguous store (scalar plus immediate)

SVE contiguous non-temporal store (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store with Immediate 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	1	1	0	0	1	0	msz		0	0	1	imm4				1	1	1	Pg			Rn				Zt					

Decode fields msz	Instruction Details
00	STNT1B (scalar plus immediate)
01	STNT1H (scalar plus immediate)
10	STNT1W (scalar plus immediate)
11	STNT1D (scalar plus immediate)

SVE store multiple structures (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store with Immediate 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 1 1 0 0 1 0							msz		!= 00		1	imm4				1 1 1			Pg			Rn				Zt					
opc																															

The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields msz	opc	Instruction Details
00	01	ST2B (scalar plus immediate)
00	10	ST3B (scalar plus immediate)
00	11	ST4B (scalar plus immediate)

Decode fields		Instruction Details
msz	opc	
01	01	ST2H (scalar plus immediate)
01	10	ST3H (scalar plus immediate)
01	11	ST4H (scalar plus immediate)
10	01	ST2W (scalar plus immediate)
10	10	ST3W (scalar plus immediate)
10	11	ST4W (scalar plus immediate)
11	01	ST2D (scalar plus immediate)
11	10	ST3D (scalar plus immediate)
11	11	ST4D (scalar plus immediate)

SVE contiguous store (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store with Immediate 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	1	1	0	0	1	0	msz		size		0	imm4				1	1	1	Pg			Rn				Zt					

Decode fields		Instruction Details
msz		
00		ST1B (scalar plus immediate)
01		ST1H (scalar plus immediate)
10		ST1W (scalar plus immediate)
11		ST1D (scalar plus immediate)

Data Processing -- Immediate

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			100			op0																									

Decode fields		Instruction details
op0		
00x		PC-rel. addressing
010		Add/subtract (immediate)
011		Add/subtract (immediate, with tags)
100		Logical (immediate)
101		Move wide (immediate)
110		Bitfield
111		Extract

PC-rel. addressing

These instructions are under [Data Processing -- Immediate](#).

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

Decode fields		Instruction Details
op		
0		ADR
1		ADRP

Add/subtract (immediate)

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
sf		op		S		1		0		0		0		1		0		sh		imm12												Rn				Rd			

Decode fields			Instruction Details
sf	op	S	
0	0	0	ADD (immediate) — 32-bit
0	0	1	ADDS (immediate) — 32-bit
0	1	0	SUB (immediate) — 32-bit
0	1	1	SUBS (immediate) — 32-bit
1	0	0	ADD (immediate) — 64-bit
1	0	1	ADDS (immediate) — 64-bit
1	1	0	SUB (immediate) — 64-bit
1	1	1	SUBS (immediate) — 64-bit

Add/subtract (immediate, with tags)

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	0	0	0	1	1	o2	uimm6						op3	uimm4				Rn				Rd						

Decode fields				Instruction Details	Feature
sf	op	S	o2		
			1	UNALLOCATED	-
0			0	UNALLOCATED	-
1		1	0	UNALLOCATED	-
1	0	0	0	ADDG	FEAT_MTE
1	1	0	0	SUBG	FEAT_MTE

Logical (immediate)

These instructions are under [Data Processing -- Immediate](#).

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

Decode fields			Instruction Details
sf	opc	N	
0		1	UNALLOCATED
0	00	0	AND (immediate) — 32-bit
0	01	0	ORR (immediate) — 32-bit
0	10	0	EOR (immediate) — 32-bit
0	11	0	ANDS (immediate) — 32-bit
1	00		AND (immediate) — 64-bit
1	01		ORR (immediate) — 64-bit
1	10		EOR (immediate) — 64-bit
1	11		ANDS (immediate) — 64-bit

Move wide (immediate)

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc	1	0	0	1	0	1	hw	imm16																						Rd

Decode fields			Instruction Details
sf	opc	hw	
	01		UNALLOCATED
0		1x	UNALLOCATED
0	00	0x	MOVN — 32-bit
0	10	0x	MOVZ — 32-bit
0	11	0x	MOVK — 32-bit
1	00		MOVN — 64-bit
1	10		MOVZ — 64-bit
1	11		MOVK — 64-bit

Bitfield

These instructions are under [Data Processing -- Immediate](#).

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

Decode fields			Instruction Details
sf	opc	N	
	11		UNALLOCATED
0		1	UNALLOCATED
0	00	0	SBFM — 32-bit
0	01	0	BFM — 32-bit
0	10	0	UBFM — 32-bit
1		0	UNALLOCATED
1	00	1	SBFM — 64-bit
1	01	1	BFM — 64-bit
1	10	1	UBFM — 64-bit

Extract

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op21	1	0	0	1	1	1	N	o0	Rm				imms				Rn				Rd									

Decode fields					Instruction Details
sf	op21	N	o0	imms	
	x1				UNALLOCATED
	00		1		UNALLOCATED
	1x				UNALLOCATED
0				1xxxxx	UNALLOCATED
0		1			UNALLOCATED
0	00	0	0	0xxxxx	EXTR — 32-bit
1		0			UNALLOCATED
1	00	1	0		EXTR — 64-bit

Branches, Exception Generating and System instructions

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0			101			op1																								op2	

Decode fields			Instruction details			
op0	op1	op2				
010	0xxxxxxxxxxxxxx		Conditional branch (immediate)			
110	00xxxxxxxxxxxxxx		Exception generation			
110	01000000110001		System instructions with register argument			
110	01000000110010	11111	Hints			
110	01000000110011		Barriers			
110	0100000xxx0100		PSTATE			
110	0100x01xxxxxxxxx		System instructions			
110	0100x1xxxxxxxxxx		System register move			
110	1xxxxxxxxxxxxxxx		Unconditional branch (register)			
x00			Unconditional branch (immediate)			
x01	0xxxxxxxxxxxxxxx		Compare and branch (immediate)			
x01	1xxxxxxxxxxxxxxx		Test and branch (immediate)			

Conditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	01	imm19																			00	cond			

Decode fields		Instruction Details
o1	o0	
0	0	B.cond
0	1	UNALLOCATED
1		UNALLOCATED

Exception generation

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	opc				imm16												op2		LL					

Decode fields			Instruction Details
opc	op2	LL	
	001		UNALLOCATED
	01x		UNALLOCATED
	1xx		UNALLOCATED
000	000	00	UNALLOCATED
000	000	01	SVC
000	000	10	HVC
000	000	11	SMC
001	000	x1	UNALLOCATED
001	000	00	BRK
001	000	1x	UNALLOCATED
010	000	x1	UNALLOCATED
010	000	00	HLT
010	000	1x	UNALLOCATED
011	000	01	UNALLOCATED

Decode fields			Instruction Details
opc	op2	LL	
011	000	1x	UNALLOCATED
100	000		UNALLOCATED
101	000	00	UNALLOCATED
101	000	01	DCPS1
101	000	10	DCPS2
101	000	11	DCPS3
110	000		UNALLOCATED
111	000		UNALLOCATED

System instructions with register argument

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	0	1	CRm			op2			Rt					

Decode fields		Instruction Details	Feature
CRm	op2		
!= 0000		UNALLOCATED	-
0000	000	WFET	FEAT_WFxT
0000	001	WFIT	FEAT_WFxT
0000	01x	UNALLOCATED	-
0000	1xx	UNALLOCATED	-

Hints

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	CRm			op2			1	1	1	1	1	

Decode fields		Instruction Details	Feature
CRm	op2		
		HINT	-
0000	000	NOP	-
0000	001	YIELD	-
0000	010	WFE	-
0000	011	WFI	-
0000	100	SEV	-
0000	101	SEVL	-
0000	110	DGH	FEAT_DGH
0000	111	XPACD, XPACI, XPACLRI	FEAT_PAuth
0001	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA1716	FEAT_PAuth
0001	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB1716	FEAT_PAuth
0001	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA1716	FEAT_PAuth
0001	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB1716	FEAT_PAuth
0010	000	ESB	FEAT_RAS
0010	001	PSB CSYNC	FEAT_SPE
0010	010	TSB CSYNC	FEAT_TRF
0010	100	CSDB	-
0011	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIAZ	FEAT_PAuth

Decode fields		Instruction Details	Feature
CRm	op2		
0011	001	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIASP	FEAT_PAuth
0011	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBZ	FEAT_PAuth
0011	011	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBSP	FEAT_PAuth
0011	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIAZ	FEAT_PAuth
0011	101	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIASP	FEAT_PAuth
0011	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBZ	FEAT_PAuth
0011	111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBSP	FEAT_PAuth
0100	xx0	BTI	FEAT_BTI

Barriers

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm			op2			Rt					

Decode fields			Instruction Details	Feature
CRm	op2	Rt		
	000		UNALLOCATED	-
	001	!= 11111	UNALLOCATED	-
	010	11111	CLREX	-
	100	11111	DSB — memory barrier	-
	101	11111	DMB	-
	110	11111	ISB	-
	111	!= 11111	UNALLOCATED	-
	111	11111	SB	-
xx0x	001	11111	UNALLOCATED	-
xx10	001	11111	DSB — Memory nXS barrier	FEAT_XS
xx11	001	11111	UNALLOCATED	-
0001	011		UNALLOCATED	-
001x	011		UNALLOCATED	-
01xx	011		UNALLOCATED	-
1xxx	011		UNALLOCATED	-

PSTATE

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	op1			0			1	0	0	CRm			op2			Rt			

Decode fields			Instruction Details	Feature
op1	op2	Rt		
		!= 11111	UNALLOCATED	-
		11111	MSR (immediate)	-
000	000	11111	CFINV	FEAT_FlagM
000	001	11111	XAFLAG	FEAT_FlagM2
000	010	11111	AXFLAG	FEAT_FlagM2

System instructions

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	L	0	1	op1			CRn			CRm			op2			Rt						

Decode fields	Instruction Details
L	
0	SYS
1	SYSL

System register move

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	L	1	o0	op1			CRn			CRm			op2			Rt						

Decode fields	Instruction Details
L	
0	MSR (register)
1	MRS

Unconditional branch (register)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	opc			op2			op3			Rn			op4												

opc	op2	Decode fields op3	Rn	op4	Instruction Details	Feature
	!= 11111				UNALLOCATED	-
0000	11111	0000000		!= 00000	UNALLOCATED	-
0000	11111	0000000		00000	BR	-
0000	11111	0000001			UNALLOCATED	-
0000	11111	0000010		!= 11111	UNALLOCATED	-
0000	11111	0000010		11111	BRAA, BRAAZ, BRAB, BRABZ — key A, zero modifier	FEAT_PAuth
0000	11111	0000011		!= 11111	UNALLOCATED	-
0000	11111	0000011		11111	BRAA, BRAAZ, BRAB, BRABZ — key B, zero modifier	FEAT_PAuth
0000	11111	0001xx			UNALLOCATED	-
0000	11111	001xxx			UNALLOCATED	-
0000	11111	01xxxx			UNALLOCATED	-
0000	11111	1xxxxx			UNALLOCATED	-
0001	11111	0000000		!= 00000	UNALLOCATED	-
0001	11111	0000000		00000	BLR	-
0001	11111	0000001			UNALLOCATED	-

Decode fields					Instruction Details	Feature
opc	op2	op3	Rn	op4		
0001	11111	000010		!= 11111	UNALLOCATED	-
0001	11111	000010		11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, zero modifier	FEAT_PAuth
0001	11111	000011		!= 11111	UNALLOCATED	-
0001	11111	000011		11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, zero modifier	FEAT_PAuth
0001	11111	0001xx			UNALLOCATED	-
0001	11111	001xxx			UNALLOCATED	-
0001	11111	01xxxx			UNALLOCATED	-
0001	11111	1xxxxx			UNALLOCATED	-
0010	11111	000000		!= 00000	UNALLOCATED	-
0010	11111	000000		00000	RET	-
0010	11111	000001			UNALLOCATED	-
0010	11111	000010	!= 11111	!= 11111	UNALLOCATED	-
0010	11111	000010	11111	11111	RETAA, RETAB — RETAA	FEAT_PAuth
0010	11111	000011	!= 11111	!= 11111	UNALLOCATED	-
0010	11111	000011	11111	11111	RETAA, RETAB — RETAB	FEAT_PAuth
0010	11111	0001xx			UNALLOCATED	-
0010	11111	001xxx			UNALLOCATED	-
0010	11111	01xxxx			UNALLOCATED	-
0010	11111	1xxxxx			UNALLOCATED	-
0011	11111				UNALLOCATED	-
0100	11111	000000	!= 11111	!= 00000	UNALLOCATED	-
0100	11111	000000	!= 11111	00000	UNALLOCATED	-
0100	11111	000000	11111	!= 00000	UNALLOCATED	-
0100	11111	000000	11111	00000	ERET	-
0100	11111	000001			UNALLOCATED	-
0100	11111	000010	!= 11111	!= 11111	UNALLOCATED	-
0100	11111	000010	!= 11111	11111	UNALLOCATED	-
0100	11111	000010	11111	!= 11111	UNALLOCATED	-
0100	11111	000010	11111	11111	ERETAA, ERETAB — ERETAA	FEAT_PAuth
0100	11111	000011	!= 11111	!= 11111	UNALLOCATED	-
0100	11111	000011	!= 11111	11111	UNALLOCATED	-
0100	11111	000011	11111	!= 11111	UNALLOCATED	-

Decode fields					Instruction Details	Feature
opc	op2	op3	Rn	op4		
0100	11111	000011	11111	11111	ERETAA, ERETAB — ERETAB	FEAT_PAuth
0100	11111	0001xx			UNALLOCATED	-
0100	11111	001xxx			UNALLOCATED	-
0100	11111	01xxxx			UNALLOCATED	-
0100	11111	1xxxxx			UNALLOCATED	-
0101	11111	!= 000000			UNALLOCATED	-
0101	11111	000000	!= 11111	!= 00000	UNALLOCATED	-
0101	11111	000000	!= 11111	00000	UNALLOCATED	-
0101	11111	000000	11111	!= 00000	UNALLOCATED	-
0101	11111	000000	11111	00000	DRPS	-
011x	11111				UNALLOCATED	-
1000	11111	00000x			UNALLOCATED	-
1000	11111	000010			BRAA, BRAAZ, BRAB, BRABZ — key A, register modifier	FEAT_PAuth
1000	11111	000011			BRAA, BRAAZ, BRAB, BRABZ — key B, register modifier	FEAT_PAuth
1000	11111	0001xx			UNALLOCATED	-
1000	11111	001xxx			UNALLOCATED	-
1000	11111	01xxxx			UNALLOCATED	-
1000	11111	1xxxxx			UNALLOCATED	-
1001	11111	00000x			UNALLOCATED	-
1001	11111	000010			BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, register modifier	FEAT_PAuth
1001	11111	000011			BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, register modifier	FEAT_PAuth
1001	11111	0001xx			UNALLOCATED	-
1001	11111	001xxx			UNALLOCATED	-
1001	11111	01xxxx			UNALLOCATED	-
1001	11111	1xxxxx			UNALLOCATED	-
101x	11111				UNALLOCATED	-
11xx	11111				UNALLOCATED	-

Unconditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
op	0	0	1	0	1	imm26																										

Decode fields		Instruction Details
op		
0		B
1		BL

Compare and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf								imm19																	Rt						

Decode fields		Instruction Details
sf	op	
0	0	CBZ — 32-bit
0	1	CBNZ — 32-bit
1	0	CBZ — 64-bit
1	1	CBNZ — 64-bit

Test and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
b5								b40				imm14																Rt			

Decode fields		Instruction Details
op		
0		TBZ
1		TBNZ

Loads and Stores

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0				1	op1	0	op2	op3				op4																			

Decode fields					Instruction details
op0	op1	op2	op3	op4	
0x00	0	00	1xxxxx		Compare and swap pair
0x00	1	00	000000		Advanced SIMD load/store multiple structures
0x00	1	01	0xxxxx		Advanced SIMD load/store multiple structures (post-indexed)
0x00	1	0x	1xxxxx		UNALLOCATED
0x00	1	10	x00000		Advanced SIMD load/store single structure
0x00	1	11			Advanced SIMD load/store single structure (post-indexed)
0x00	1	x0	x1xxxx		UNALLOCATED
0x00	1	x0	xx1xxx		UNALLOCATED
0x00	1	x0	xxx1xx		UNALLOCATED
0x00	1	x0	xxxx1x		UNALLOCATED
0x00	1	x0	xxxxx1		UNALLOCATED
1101	0	1x	1xxxxx		Load/store memory tags
1x00	0	00	1xxxxx		Load/store exclusive pair
1x00	1				UNALLOCATED
xx00	0	00	0xxxxx		Load/store exclusive register
xx00	0	01	0xxxxx		Load/store ordered
xx00	0	01	1xxxxx		Compare and swap
xx01	0	1x	0xxxxx	00	LDAPR/STLR (unscaled immediate)
xx01		0x			Load register (literal)
xx10		00			Load/store no-allocate pair (offset)
xx10		01			Load/store register pair (post-indexed)
xx10		10			Load/store register pair (offset)
xx10		11			Load/store register pair (pre-indexed)

xx11		0x	0xxxxx	00	Load/store register (unscaled immediate)
xx11		0x	0xxxxx	01	Load/store register (immediate post-indexed)
xx11		0x	0xxxxx	10	Load/store register (unprivileged)
xx11		0x	0xxxxx	11	Load/store register (immediate pre-indexed)
xx11		0x	1xxxxx	00	Atomic memory operations
xx11		0x	1xxxxx	10	Load/store register (register offset)
xx11		0x	1xxxxx	x1	Load/store register (pac)
xx11		1x			Load/store register (unsigned immediate)

Compare and swap pair

These instructions are under [Loads and Stores](#).

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

Decode fields				Instruction Details		Feature
sz	L	o0	Rt2			
			!= 11111	UNALLOCATED		-
0	0	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASP		FEAT_LSE
0	0	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPL		FEAT_LSE
0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPA		FEAT_LSE
0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPAL		FEAT_LSE
1	0	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASP		FEAT_LSE
1	0	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPL		FEAT_LSE
1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPA		FEAT_LSE
1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL		FEAT_LSE

Advanced SIMD load/store multiple structures

These instructions are under [Loads and Stores](#).

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

Decode fields		Instruction Details	
L	opcode		
0	0000	ST4 (multiple structures)	
0	0001	UNALLOCATED	
0	0010	ST1 (multiple structures) — four registers	
0	0011	UNALLOCATED	
0	0100	ST3 (multiple structures)	
0	0101	UNALLOCATED	
0	0110	ST1 (multiple structures) — three registers	
0	0111	ST1 (multiple structures) — one register	
0	1000	ST2 (multiple structures)	
0	1001	UNALLOCATED	
0	1010	ST1 (multiple structures) — two registers	
0	1011	UNALLOCATED	
0	11xx	UNALLOCATED	
1	0000	LD4 (multiple structures)	
1	0001	UNALLOCATED	

Decode fields		Instruction Details
L	opcode	
1	0010	LD1 (multiple structures) — four registers
1	0011	UNALLOCATED
1	0100	LD3 (multiple structures)
1	0101	UNALLOCATED
1	0110	LD1 (multiple structures) — three registers
1	0111	LD1 (multiple structures) — one register
1	1000	LD2 (multiple structures)
1	1001	UNALLOCATED
1	1010	LD1 (multiple structures) — two registers
1	1011	UNALLOCATED
1	11xx	UNALLOCATED

Advanced SIMD load/store multiple structures (post-indexed)

These instructions are under [Loads and Stores](#).

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

Decode fields		Instruction Details
L	Rm opcode	
0		0001 UNALLOCATED
0		0011 UNALLOCATED
0		0101 UNALLOCATED
0		1001 UNALLOCATED
0		1011 UNALLOCATED
0		11xx UNALLOCATED
0	!= 11111	0000 ST4 (multiple structures) — register offset
0	!= 11111	0010 ST1 (multiple structures) — four registers, register offset
0	!= 11111	0100 ST3 (multiple structures) — register offset
0	!= 11111	0110 ST1 (multiple structures) — three registers, register offset
0	!= 11111	0111 ST1 (multiple structures) — one register, register offset
0	!= 11111	1000 ST2 (multiple structures) — register offset
0	!= 11111	1010 ST1 (multiple structures) — two registers, register offset
0	11111	0000 ST4 (multiple structures) — immediate offset
0	11111	0010 ST1 (multiple structures) — four registers, immediate offset
0	11111	0100 ST3 (multiple structures) — immediate offset
0	11111	0110 ST1 (multiple structures) — three registers, immediate offset
0	11111	0111 ST1 (multiple structures) — one register, immediate offset
0	11111	1000 ST2 (multiple structures) — immediate offset
0	11111	1010 ST1 (multiple structures) — two registers, immediate offset
1		0001 UNALLOCATED
1		0011 UNALLOCATED
1		0101 UNALLOCATED
1		1001 UNALLOCATED
1		1011 UNALLOCATED
1		11xx UNALLOCATED
1	!= 11111	0000 LD4 (multiple structures) — register offset
1	!= 11111	0010 LD1 (multiple structures) — four registers, register offset

Decode fields			Instruction Details
L	Rm	opcode	
1	!= 11111	0100	LD3 (multiple structures) — register offset
1	!= 11111	0110	LD1 (multiple structures) — three registers, register offset
1	!= 11111	0111	LD1 (multiple structures) — one register, register offset
1	!= 11111	1000	LD2 (multiple structures) — register offset
1	!= 11111	1010	LD1 (multiple structures) — two registers, register offset
1	11111	0000	LD4 (multiple structures) — immediate offset
1	11111	0010	LD1 (multiple structures) — four registers, immediate offset
1	11111	0100	LD3 (multiple structures) — immediate offset
1	11111	0110	LD1 (multiple structures) — three registers, immediate offset
1	11111	0111	LD1 (multiple structures) — one register, immediate offset
1	11111	1000	LD2 (multiple structures) — immediate offset
1	11111	1010	LD1 (multiple structures) — two registers, immediate offset

Advanced SIMD load/store single structure

These instructions are under [Loads and Stores](#).

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

Decode fields					Instruction Details
L	R	opcode	S	size	
0		11x			UNALLOCATED
0	0	000			ST1 (single structure) — 8-bit
0	0	001			ST3 (single structure) — 8-bit
0	0	010		x0	ST1 (single structure) — 16-bit
0	0	010		x1	UNALLOCATED
0	0	011		x0	ST3 (single structure) — 16-bit
0	0	011		x1	UNALLOCATED
0	0	100		00	ST1 (single structure) — 32-bit
0	0	100		1x	UNALLOCATED
0	0	100	0	01	ST1 (single structure) — 64-bit
0	0	100	1	01	UNALLOCATED
0	0	101		00	ST3 (single structure) — 32-bit
0	0	101		10	UNALLOCATED
0	0	101	0	01	ST3 (single structure) — 64-bit
0	0	101	0	11	UNALLOCATED
0	0	101	1	x1	UNALLOCATED
0	1	000			ST2 (single structure) — 8-bit
0	1	001			ST4 (single structure) — 8-bit
0	1	010		x0	ST2 (single structure) — 16-bit
0	1	010		x1	UNALLOCATED
0	1	011		x0	ST4 (single structure) — 16-bit
0	1	011		x1	UNALLOCATED
0	1	100		00	ST2 (single structure) — 32-bit
0	1	100		10	UNALLOCATED
0	1	100	0	01	ST2 (single structure) — 64-bit
0	1	100	0	11	UNALLOCATED
0	1	100	1	x1	UNALLOCATED

Decode fields					Instruction Details
L	R	opcode	S	size	
0	1	101		00	ST4 (single structure) — 32-bit
0	1	101		10	UNALLOCATED
0	1	101	0	01	ST4 (single structure) — 64-bit
0	1	101	0	11	UNALLOCATED
0	1	101	1	x1	UNALLOCATED
1	0	000			LD1 (single structure) — 8-bit
1	0	001			LD3 (single structure) — 8-bit
1	0	010		x0	LD1 (single structure) — 16-bit
1	0	010		x1	UNALLOCATED
1	0	011		x0	LD3 (single structure) — 16-bit
1	0	011		x1	UNALLOCATED
1	0	100		00	LD1 (single structure) — 32-bit
1	0	100		1x	UNALLOCATED
1	0	100	0	01	LD1 (single structure) — 64-bit
1	0	100	1	01	UNALLOCATED
1	0	101		00	LD3 (single structure) — 32-bit
1	0	101		10	UNALLOCATED
1	0	101	0	01	LD3 (single structure) — 64-bit
1	0	101	0	11	UNALLOCATED
1	0	101	1	x1	UNALLOCATED
1	0	110	0		LD1R
1	0	110	1		UNALLOCATED
1	0	111	0		LD3R
1	0	111	1		UNALLOCATED
1	1	000			LD2 (single structure) — 8-bit
1	1	001			LD4 (single structure) — 8-bit
1	1	010		x0	LD2 (single structure) — 16-bit
1	1	010		x1	UNALLOCATED
1	1	011		x0	LD4 (single structure) — 16-bit
1	1	011		x1	UNALLOCATED
1	1	100		00	LD2 (single structure) — 32-bit
1	1	100		10	UNALLOCATED
1	1	100	0	01	LD2 (single structure) — 64-bit
1	1	100	0	11	UNALLOCATED
1	1	100	1	x1	UNALLOCATED
1	1	101		00	LD4 (single structure) — 32-bit
1	1	101		10	UNALLOCATED
1	1	101	0	01	LD4 (single structure) — 64-bit
1	1	101	0	11	UNALLOCATED
1	1	101	1	x1	UNALLOCATED
1	1	110	0		LD2R
1	1	110	1		UNALLOCATED
1	1	111	0		LD4R
1	1	111	1		UNALLOCATED

Advanced SIMD load/store single structure (post-indexed)

These instructions are under [Loads and Stores](#).

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

Decode fields						Instruction Details
L	R	Rm	opcode	S	size	
0			11x			UNALLOCATED
0	0		010		x1	UNALLOCATED
0	0		011		x1	UNALLOCATED
0	0		100		1x	UNALLOCATED
0	0		100	1	01	UNALLOCATED
0	0		101		10	UNALLOCATED
0	0		101	0	11	UNALLOCATED
0	0		101	1	x1	UNALLOCATED
0	0	!= 11111	000			ST1 (single structure) — 8-bit, register offset
0	0	!= 11111	001			ST3 (single structure) — 8-bit, register offset
0	0	!= 11111	010		x0	ST1 (single structure) — 16-bit, register offset
0	0	!= 11111	011		x0	ST3 (single structure) — 16-bit, register offset
0	0	!= 11111	100		00	ST1 (single structure) — 32-bit, register offset
0	0	!= 11111	100	0	01	ST1 (single structure) — 64-bit, register offset
0	0	!= 11111	101		00	ST3 (single structure) — 32-bit, register offset
0	0	!= 11111	101	0	01	ST3 (single structure) — 64-bit, register offset
0	0	11111	000			ST1 (single structure) — 8-bit, immediate offset
0	0	11111	001			ST3 (single structure) — 8-bit, immediate offset
0	0	11111	010		x0	ST1 (single structure) — 16-bit, immediate offset
0	0	11111	011		x0	ST3 (single structure) — 16-bit, immediate offset
0	0	11111	100		00	ST1 (single structure) — 32-bit, immediate offset
0	0	11111	100	0	01	ST1 (single structure) — 64-bit, immediate offset
0	0	11111	101		00	ST3 (single structure) — 32-bit, immediate offset
0	0	11111	101	0	01	ST3 (single structure) — 64-bit, immediate offset
0	1		010		x1	UNALLOCATED
0	1		011		x1	UNALLOCATED
0	1		100		10	UNALLOCATED
0	1		100	0	11	UNALLOCATED
0	1		100	1	x1	UNALLOCATED
0	1		101		10	UNALLOCATED
0	1		101	0	11	UNALLOCATED
0	1		101	1	x1	UNALLOCATED
0	1	!= 11111	000			ST2 (single structure) — 8-bit, register offset
0	1	!= 11111	001			ST4 (single structure) — 8-bit, register offset
0	1	!= 11111	010		x0	ST2 (single structure) — 16-bit, register offset
0	1	!= 11111	011		x0	ST4 (single structure) — 16-bit, register offset
0	1	!= 11111	100		00	ST2 (single structure) — 32-bit, register offset
0	1	!= 11111	100	0	01	ST2 (single structure) — 64-bit, register offset
0	1	!= 11111	101		00	ST4 (single structure) — 32-bit, register offset
0	1	!= 11111	101	0	01	ST4 (single structure) — 64-bit, register offset
0	1	11111	000			ST2 (single structure) — 8-bit, immediate offset
0	1	11111	001			ST4 (single structure) — 8-bit, immediate offset
0	1	11111	010		x0	ST2 (single structure) — 16-bit, immediate offset
0	1	11111	011		x0	ST4 (single structure) — 16-bit, immediate offset
0	1	11111	100		00	ST2 (single structure) — 32-bit, immediate offset
0	1	11111	100	0	01	ST2 (single structure) — 64-bit, immediate offset

L	R	Decode fields		S	size	Instruction Details
		Rm	opcode			
0	1	11111	101		00	ST4 (single structure) — 32-bit, immediate offset
0	1	11111	101	0	01	ST4 (single structure) — 64-bit, immediate offset
1	0		010		x1	UNALLOCATED
1	0		011		x1	UNALLOCATED
1	0		100		1x	UNALLOCATED
1	0		100	1	01	UNALLOCATED
1	0		101		10	UNALLOCATED
1	0		101	0	11	UNALLOCATED
1	0		101	1	x1	UNALLOCATED
1	0		110	1		UNALLOCATED
1	0		111	1		UNALLOCATED
1	0	!= 11111	000			LD1 (single structure) — 8-bit, register offset
1	0	!= 11111	001			LD3 (single structure) — 8-bit, register offset
1	0	!= 11111	010		x0	LD1 (single structure) — 16-bit, register offset
1	0	!= 11111	011		x0	LD3 (single structure) — 16-bit, register offset
1	0	!= 11111	100		00	LD1 (single structure) — 32-bit, register offset
1	0	!= 11111	100	0	01	LD1 (single structure) — 64-bit, register offset
1	0	!= 11111	101		00	LD3 (single structure) — 32-bit, register offset
1	0	!= 11111	101	0	01	LD3 (single structure) — 64-bit, register offset
1	0	!= 11111	110	0		LD1R — register offset
1	0	!= 11111	111	0		LD3R — register offset
1	0	11111	000			LD1 (single structure) — 8-bit, immediate offset
1	0	11111	001			LD3 (single structure) — 8-bit, immediate offset
1	0	11111	010		x0	LD1 (single structure) — 16-bit, immediate offset
1	0	11111	011		x0	LD3 (single structure) — 16-bit, immediate offset
1	0	11111	100		00	LD1 (single structure) — 32-bit, immediate offset
1	0	11111	100	0	01	LD1 (single structure) — 64-bit, immediate offset
1	0	11111	101		00	LD3 (single structure) — 32-bit, immediate offset
1	0	11111	101	0	01	LD3 (single structure) — 64-bit, immediate offset
1	0	11111	110	0		LD1R — immediate offset
1	0	11111	111	0		LD3R — immediate offset
1	1		010		x1	UNALLOCATED
1	1		011		x1	UNALLOCATED
1	1		100		10	UNALLOCATED
1	1		100	0	11	UNALLOCATED
1	1		100	1	x1	UNALLOCATED
1	1		101		10	UNALLOCATED
1	1		101	0	11	UNALLOCATED
1	1		101	1	x1	UNALLOCATED
1	1		110	1		UNALLOCATED
1	1		111	1		UNALLOCATED
1	1	!= 11111	000			LD2 (single structure) — 8-bit, register offset
1	1	!= 11111	001			LD4 (single structure) — 8-bit, register offset
1	1	!= 11111	010		x0	LD2 (single structure) — 16-bit, register offset
1	1	!= 11111	011		x0	LD4 (single structure) — 16-bit, register offset
1	1	!= 11111	100		00	LD2 (single structure) — 32-bit, register offset
1	1	!= 11111	100	0	01	LD2 (single structure) — 64-bit, register offset

Decode fields						Instruction Details
L	R	Rm	opcode	S	size	
1	1	!= 11111	101		00	LD4 (single structure) — 32-bit, register offset
1	1	!= 11111	101	0	01	LD4 (single structure) — 64-bit, register offset
1	1	!= 11111	110	0		LD2R — register offset
1	1	!= 11111	111	0		LD4R — register offset
1	1	11111	000			LD2 (single structure) — 8-bit, immediate offset
1	1	11111	001			LD4 (single structure) — 8-bit, immediate offset
1	1	11111	010		x0	LD2 (single structure) — 16-bit, immediate offset
1	1	11111	011		x0	LD4 (single structure) — 16-bit, immediate offset
1	1	11111	100		00	LD2 (single structure) — 32-bit, immediate offset
1	1	11111	100	0	01	LD2 (single structure) — 64-bit, immediate offset
1	1	11111	101		00	LD4 (single structure) — 32-bit, immediate offset
1	1	11111	101	0	01	LD4 (single structure) — 64-bit, immediate offset
1	1	11111	110	0		LD2R — immediate offset
1	1	11111	111	0		LD4R — immediate offset

Load/store memory tags

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	opc	1	imm9									op2	Rn				Rt							

opc	Decode fields imm9	op2	Instruction Details	Feature
00		01	STG — post-index	FEAT_MTE
00		10	STG — signed offset	FEAT_MTE
00		11	STG — pre-index	FEAT_MTE
00	000000000	00	STZGM	FEAT_MTE2
01		00	LDG	FEAT_MTE
01		01	STZG — post-index	FEAT_MTE
01		10	STZG — signed offset	FEAT_MTE
01		11	STZG — pre-index	FEAT_MTE
10		01	ST2G — post-index	FEAT_MTE
10		10	ST2G — signed offset	FEAT_MTE
10		11	ST2G — pre-index	FEAT_MTE
10	!= 000000000	00	UNALLOCATED	-
10	000000000	00	STGM	FEAT_MTE2
11		01	STZ2G — post-index	FEAT_MTE
11		10	STZ2G — signed offset	FEAT_MTE
11		11	STZ2G — pre-index	FEAT_MTE
11	!= 000000000	00	UNALLOCATED	-
11	000000000	00	LDGM	FEAT_MTE2

Load/store exclusive pair

These instructions are under [Loads and Stores](#).

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

Decode fields			Instruction Details
sz	L	o0	
0	0	0	STXP — 32-bit
0	0	1	STLXP — 32-bit
0	1	0	LDXP — 32-bit
0	1	1	LDAXP — 32-bit
1	0	0	STXP — 64-bit
1	0	1	STLXP — 64-bit
1	1	0	LDXP — 64-bit
1	1	1	LDAXP — 64-bit

Load/store exclusive register

These instructions are under [Loads and Stores](#).

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

Decode fields			Instruction Details
size	L	o0	
00	0	0	STXRB
00	0	1	STLXRB
00	1	0	LDXRB
00	1	1	LDAXRB
01	0	0	STXRH
01	0	1	STLXRH
01	1	0	LDXRH
01	1	1	LDAXRH
10	0	0	STXR — 32-bit
10	0	1	STLXR — 32-bit
10	1	0	LDXR — 32-bit
10	1	1	LDAXR — 32-bit
11	0	0	STXR — 64-bit
11	0	1	STLXR — 64-bit
11	1	0	LDXR — 64-bit
11	1	1	LDAXR — 64-bit

Load/store ordered

These instructions are under [Loads and Stores](#).

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

Decode fields			Instruction Details	Feature
size	L	o0		
00	0	0	STLLRB	FEAT_LOR
00	0	1	STLRB	-
00	1	0	LDLARB	FEAT_LOR
00	1	1	LDARB	-
01	0	0	STLLRH	FEAT_LOR
01	0	1	STLRH	-

Decode fields size	L	o0	Instruction Details	Feature
01	1	0	LDLARH	FEAT_LOR
01	1	1	LDARH	-
10	0	0	STLLR — 32-bit	FEAT_LOR
10	0	1	STLR — 32-bit	-
10	1	0	LDLAR — 32-bit	FEAT_LOR
10	1	1	LDAR — 32-bit	-
11	0	0	STLLR — 64-bit	FEAT_LOR
11	0	1	STLR — 64-bit	-
11	1	0	LDLAR — 64-bit	FEAT_LOR
11	1	1	LDAR — 64-bit	-

Compare and swap

These instructions are under [Loads and Stores](#).

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

Decode fields size	L	o0	Rt2	Instruction Details	Feature
			!= 11111	UNALLOCATED	-
00	0	0	11111	CASB, CASAB, CASALB, CASLB — CASB	FEAT_LSE
00	0	1	11111	CASB, CASAB, CASALB, CASLB — CASLB	FEAT_LSE
00	1	0	11111	CASB, CASAB, CASALB, CASLB — CASAB	FEAT_LSE
00	1	1	11111	CASB, CASAB, CASALB, CASLB — CASALB	FEAT_LSE
01	0	0	11111	CASH, CASAH, CASALH, CASLH — CASH	FEAT_LSE
01	0	1	11111	CASH, CASAH, CASALH, CASLH — CASLH	FEAT_LSE
01	1	0	11111	CASH, CASAH, CASALH, CASLH — CASAH	FEAT_LSE
01	1	1	11111	CASH, CASAH, CASALH, CASLH — CASALH	FEAT_LSE
10	0	0	11111	CAS, CASA, CASAL, CASL — 32-bit CAS	FEAT_LSE
10	0	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASL	FEAT_LSE
10	1	0	11111	CAS, CASA, CASAL, CASL — 32-bit CASA	FEAT_LSE
10	1	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASAL	FEAT_LSE
11	0	0	11111	CAS, CASA, CASAL, CASL — 64-bit CAS	FEAT_LSE
11	0	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASL	FEAT_LSE
11	1	0	11111	CAS, CASA, CASAL, CASL — 64-bit CASA	FEAT_LSE
11	1	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASAL	FEAT_LSE

LDAPR/STLR (unscaled immediate)

These instructions are under [Loads and Stores](#).

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

Decode fields size	opc	Instruction Details	Feature
00	00	STLURB	FEAT_LRCPC2
00	01	LDAPURB	FEAT_LRCPC2
00	10	LDAPURSB — 64-bit	FEAT_LRCPC2

Decode fields size	opc	Instruction Details	Feature
00	11	LDAPURSB — 32-bit	FEAT_LRCPC2
01	00	STLURH	FEAT_LRCPC2
01	01	LDAPURH	FEAT_LRCPC2
01	10	LDAPURSH — 64-bit	FEAT_LRCPC2
01	11	LDAPURSH — 32-bit	FEAT_LRCPC2
10	00	STLUR — 32-bit	FEAT_LRCPC2
10	01	LDAPUR — 32-bit	FEAT_LRCPC2
10	10	LDAPURSW	FEAT_LRCPC2
10	11	UNALLOCATED	-
11	00	STLUR — 64-bit	FEAT_LRCPC2
11	01	LDAPUR — 64-bit	FEAT_LRCPC2
11	10	UNALLOCATED	-
11	11	UNALLOCATED	-

Load register (literal)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		0	1	1	V	0	0	imm19																	Rt						

Decode fields opc	V	Instruction Details
00	0	LDR (literal) — 32-bit
00	1	LDR (literal, SIMD&FP) — 32-bit
01	0	LDR (literal) — 64-bit
01	1	LDR (literal, SIMD&FP) — 64-bit
10	0	LDRSW (literal)
10	1	LDR (literal, SIMD&FP) — 128-bit
11	0	PRFM (literal)
11	1	UNALLOCATED

Load/store no-allocate pair (offset)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	V	0	0	0	L	imm7							Rt2				Rn				Rt						

Decode fields opc	V	L	Instruction Details
00	0	0	STNP — 32-bit
00	0	1	LDNP — 32-bit
00	1	0	STNP (SIMD&FP) — 32-bit
00	1	1	LDNP (SIMD&FP) — 32-bit
01	0		UNALLOCATED
01	1	0	STNP (SIMD&FP) — 64-bit
01	1	1	LDNP (SIMD&FP) — 64-bit
10	0	0	STNP — 64-bit
10	0	1	LDNP — 64-bit

Decode fields			Instruction Details
opc	V	L	
10	1	0	STNP (SIMD&FP) — 128-bit
10	1	1	LDNP (SIMD&FP) — 128-bit
11			UNALLOCATED

Load/store register pair (post-indexed)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc	1	0	1	V	0	0	1	L	imm7							Rt2				Rn				Rt							

Decode fields			Instruction Details	Feature
opc	V	L		
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	-
00	1	1	LDP (SIMD&FP) — 32-bit	-
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	-
01	1	1	LDP (SIMD&FP) — 64-bit	-
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	-
10	1	1	LDP (SIMD&FP) — 128-bit	-
11			UNALLOCATED	-

Load/store register pair (offset)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc	1	0	1	V	0	1	0	L	imm7							Rt2				Rn				Rt							

Decode fields			Instruction Details	Feature
opc	V	L		
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	-
00	1	1	LDP (SIMD&FP) — 32-bit	-
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	-
01	1	1	LDP (SIMD&FP) — 64-bit	-
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	-
10	1	1	LDP (SIMD&FP) — 128-bit	-
11			UNALLOCATED	-

Load/store register pair (pre-indexed)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	V	0	1	1	L	imm7							Rt2				Rn				Rt						

Decode fields			Instruction Details	Feature
opc	V	L		
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	-
00	1	1	LDP (SIMD&FP) — 32-bit	-
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	-
01	1	1	LDP (SIMD&FP) — 64-bit	-
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	-
10	1	1	LDP (SIMD&FP) — 128-bit	-
11			UNALLOCATED	-

Load/store register (unscaled immediate)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
size		1	1	1	V	0	0	opc		0	imm9										0	0	Rn						Rt			

Decode fields			Instruction Details
size	V	opc	
x1	1	1x	UNALLOCATED
00	0	00	STURB
00	0	01	LDURB
00	0	10	LDURSB — 64-bit
00	0	11	LDURSB — 32-bit
00	1	00	STUR (SIMD&FP) — 8-bit
00	1	01	LDUR (SIMD&FP) — 8-bit
00	1	10	STUR (SIMD&FP) — 128-bit
00	1	11	LDUR (SIMD&FP) — 128-bit
01	0	00	STURH
01	0	01	LDURH
01	0	10	LDURSH — 64-bit
01	0	11	LDURSH — 32-bit
01	1	00	STUR (SIMD&FP) — 16-bit
01	1	01	LDUR (SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STUR — 32-bit
10	0	01	LDUR — 32-bit
10	0	10	LDURSW
10	1	00	STUR (SIMD&FP) — 32-bit

Decode fields			Instruction Details
size	V	opc	
10	1	01	LDUR (SIMD&FP) — 32-bit
11	0	00	STUR — 64-bit
11	0	01	LDUR — 64-bit
11	0	10	PRFUM
11	1	00	STUR (SIMD&FP) — 64-bit
11	1	01	LDUR (SIMD&FP) — 64-bit

Load/store register (immediate post-indexed)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	V	0	0	opc	0	imm9											0	1	Rn				Rt					

Decode fields			Instruction Details
size	V	opc	
x1	1	1x	UNALLOCATED
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit
00	1	10	STR (immediate, SIMD&FP) — 128-bit
00	1	11	LDR (immediate, SIMD&FP) — 128-bit
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) — 64-bit
01	0	11	LDRSH (immediate) — 32-bit
01	1	00	STR (immediate, SIMD&FP) — 16-bit
01	1	01	LDR (immediate, SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STR (immediate) — 32-bit
10	0	01	LDR (immediate) — 32-bit
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) — 32-bit
10	1	01	LDR (immediate, SIMD&FP) — 32-bit
11	0	00	STR (immediate) — 64-bit
11	0	01	LDR (immediate) — 64-bit
11	0	10	UNALLOCATED
11	1	00	STR (immediate, SIMD&FP) — 64-bit
11	1	01	LDR (immediate, SIMD&FP) — 64-bit

Load/store register (unprivileged)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	V	0	0	opc	0	imm9									1	0	Rn				Rt							

Decode fields size	V	opc	Instruction Details
	1		UNALLOCATED
00	0	00	STTRB
00	0	01	LDTRB
00	0	10	LDTRSB — 64-bit
00	0	11	LDTRSB — 32-bit
01	0	00	STTRH
01	0	01	LDTRH
01	0	10	LDTRSH — 64-bit
01	0	11	LDTRSH — 32-bit
1x	0	11	UNALLOCATED
10	0	00	STTR — 32-bit
10	0	01	LDTR — 32-bit
10	0	10	LDTRSW
11	0	00	STTR — 64-bit
11	0	01	LDTR — 64-bit
11	0	10	UNALLOCATED

Load/store register (immediate pre-indexed)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	0	opc		0	imm9									1	1	Rn				Rt					

Decode fields size	V	opc	Instruction Details
x1	1	1x	UNALLOCATED
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit
00	1	10	STR (immediate, SIMD&FP) — 128-bit
00	1	11	LDR (immediate, SIMD&FP) — 128-bit
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) — 64-bit
01	0	11	LDRSH (immediate) — 32-bit
01	1	00	STR (immediate, SIMD&FP) — 16-bit
01	1	01	LDR (immediate, SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STR (immediate) — 32-bit
10	0	01	LDR (immediate) — 32-bit
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) — 32-bit
10	1	01	LDR (immediate, SIMD&FP) — 32-bit
11	0	00	STR (immediate) — 64-bit

Decode fields			Instruction Details
size	V	opc	
11	0	01	LDR (immediate) — 64-bit
11	0	10	UNALLOCATED
11	1	00	STR (immediate, SIMD&FP) — 64-bit
11	1	01	LDR (immediate, SIMD&FP) — 64-bit

Atomic memory operations

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	0	A	R	1	Rs					o3	opc			0	0	Rn					Rt				

Decode fields							Instruction Details	Feature
size	V	A	R	Rs	o3	opc		
	0				1	11x	UNALLOCATED	-
	0	0			1	100	UNALLOCATED	-
	0	0	1		1	001	UNALLOCATED	-
	0	0	1		1	010	UNALLOCATED	-
	0	0	1		1	011	UNALLOCATED	-
	0	0	1		1	101	UNALLOCATED	-
	0	1	0		1	001	UNALLOCATED	-
	0	1	0		1	010	UNALLOCATED	-
	0	1	0		1	011	UNALLOCATED	-
	0	1	0		1	101	UNALLOCATED	-
	0	1	1		1	001	UNALLOCATED	-
	0	1	1		1	010	UNALLOCATED	-
	0	1	1		1	011	UNALLOCATED	-
	0	1	1		1	100	UNALLOCATED	-
	0	1	1		1	101	UNALLOCATED	-
	1						UNALLOCATED	-
00	0	0	0		0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDB	FEAT_LSE
00	0	0	0		0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRB	FEAT_LSE
00	0	0	0		0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORB	FEAT_LSE
00	0	0	0		0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETB	FEAT_LSE
00	0	0	0		0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXB	FEAT_LSE
00	0	0	0		0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINB	FEAT_LSE
00	0	0	0		0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXB	FEAT_LSE
00	0	0	0		0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINB	FEAT_LSE
00	0	0	0		1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPB	FEAT_LSE
00	0	0	0		1	001	UNALLOCATED	-
00	0	0	0		1	010	UNALLOCATED	-
00	0	0	0		1	011	UNALLOCATED	-
00	0	0	0		1	101	UNALLOCATED	-

size	Decode fields				Instruction Details			Feature
	V	A	R	Rs	o3	opc		
00	0	0	1		0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDLB	FEAT_LSE
00	0	0	1		0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRLB	FEAT_LSE
00	0	0	1		0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORLB	FEAT_LSE
00	0	0	1		0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETLB	FEAT_LSE
00	0	0	1		0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXLB	FEAT_LSE
00	0	0	1		0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINLB	FEAT_LSE
00	0	0	1		0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXLB	FEAT_LSE
00	0	0	1		0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINLB	FEAT_LSE
00	0	0	1		1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPLB	FEAT_LSE
00	0	1	0		0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDAB	FEAT_LSE
00	0	1	0		0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRAB	FEAT_LSE
00	0	1	0		0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORAB	FEAT_LSE
00	0	1	0		0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETAB	FEAT_LSE
00	0	1	0		0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXAB	FEAT_LSE
00	0	1	0		0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINAB	FEAT_LSE
00	0	1	0		0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXAB	FEAT_LSE
00	0	1	0		0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINAB	FEAT_LSE
00	0	1	0		1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPAB	FEAT_LSE
00	0	1	0		1	100	LDAPRB	FEAT_LRCPC
00	0	1	1		0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDALB	FEAT_LSE
00	0	1	1		0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRALB	FEAT_LSE
00	0	1	1		0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORALB	FEAT_LSE
00	0	1	1		0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETALB	FEAT_LSE
00	0	1	1		0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXALB	FEAT_LSE
00	0	1	1		0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINALB	FEAT_LSE
00	0	1	1		0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXALB	FEAT_LSE
00	0	1	1		0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINALB	FEAT_LSE
00	0	1	1		1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPALB	FEAT_LSE
01	0	0	0		0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDH	FEAT_LSE
01	0	0	0		0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRH	FEAT_LSE

size	Decode fields				o3	opc	Instruction Details	Feature
	V	A	R	Rs				
01	0	0	0		0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORH	FEAT_LSE
01	0	0	0		0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETH	FEAT_LSE
01	0	0	0		0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXH	FEAT_LSE
01	0	0	0		0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINH	FEAT_LSE
01	0	0	0		0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXH	FEAT_LSE
01	0	0	0		0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINH	FEAT_LSE
01	0	0	0		1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPH	FEAT_LSE
01	0	0	0		1	001	UNALLOCATED	-
01	0	0	0		1	010	UNALLOCATED	-
01	0	0	0		1	011	UNALLOCATED	-
01	0	0	0		1	101	UNALLOCATED	-
01	0	0	1		0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDLH	FEAT_LSE
01	0	0	1		0	001	LDCLR H, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRLH	FEAT_LSE
01	0	0	1		0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORLH	FEAT_LSE
01	0	0	1		0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETLH	FEAT_LSE
01	0	0	1		0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXLH	FEAT_LSE
01	0	0	1		0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINLH	FEAT_LSE
01	0	0	1		0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXLH	FEAT_LSE
01	0	0	1		0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINLH	FEAT_LSE
01	0	0	1		1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPLH	FEAT_LSE
01	0	1	0		0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDAH	FEAT_LSE
01	0	1	0		0	001	LDCLR H, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRAH	FEAT_LSE
01	0	1	0		0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORAH	FEAT_LSE
01	0	1	0		0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETAH	FEAT_LSE
01	0	1	0		0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXAH	FEAT_LSE
01	0	1	0		0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINAH	FEAT_LSE
01	0	1	0		0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXAH	FEAT_LSE
01	0	1	0		0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINAH	FEAT_LSE
01	0	1	0		1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPAH	FEAT_LSE
01	0	1	0		1	100	LDAPRH	FEAT_LRCPC
01	0	1	1		0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDALH	FEAT_LSE

size	Decode fields				Instruction Details			Feature
	V	A	R	Rs	o3	opc		
01	0	1	1		0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRALH	FEAT_LSE
01	0	1	1		0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORALH	FEAT_LSE
01	0	1	1		0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETALH	FEAT_LSE
01	0	1	1		0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXALH	FEAT_LSE
01	0	1	1		0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINALH	FEAT_LSE
01	0	1	1		0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXALH	FEAT_LSE
01	0	1	1		0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINALH	FEAT_LSE
01	0	1	1		1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPALH	FEAT_LSE
10	0	0	0		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADD	FEAT_LSE
10	0	0	0		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLR	FEAT_LSE
10	0	0	0		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR	FEAT_LSE
10	0	0	0		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET	FEAT_LSE
10	0	0	0		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAX	FEAT_LSE
10	0	0	0		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMIN	FEAT_LSE
10	0	0	0		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAX	FEAT_LSE
10	0	0	0		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMIN	FEAT_LSE
10	0	0	0		1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWP	FEAT_LSE
10	0	0	0		1	001	UNALLOCATED	-
10	0	0	0		1	010	UNALLOCATED	-
10	0	0	0		1	011	UNALLOCATED	-
10	0	0	0		1	101	UNALLOCATED	-
10	0	0	1		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDL	FEAT_LSE
10	0	0	1		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRL	FEAT_LSE
10	0	0	1		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORL	FEAT_LSE
10	0	0	1		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETL	FEAT_LSE
10	0	0	1		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXL	FEAT_LSE
10	0	0	1		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINL	FEAT_LSE
10	0	0	1		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXL	FEAT_LSE
10	0	0	1		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINL	FEAT_LSE
10	0	0	1		1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPL	FEAT_LSE
10	0	1	0		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDA	FEAT_LSE

size	Decode fields				Instruction Details			Feature
	V	A	R	Rs	o3	opc		
10	0	1	0		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRA	FEAT_LSE
10	0	1	0		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORA	FEAT_LSE
10	0	1	0		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETA	FEAT_LSE
10	0	1	0		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXA	FEAT_LSE
10	0	1	0		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINA	FEAT_LSE
10	0	1	0		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXA	FEAT_LSE
10	0	1	0		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINA	FEAT_LSE
10	0	1	0		1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPA	FEAT_LSE
10	0	1	0		1	100	LDAPR — 32-bit	FEAT_LRCPC
10	0	1	1		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDAL	FEAT_LSE
10	0	1	1		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRAL	FEAT_LSE
10	0	1	1		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORAL	FEAT_LSE
10	0	1	1		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETAL	FEAT_LSE
10	0	1	1		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXAL	FEAT_LSE
10	0	1	1		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINAL	FEAT_LSE
10	0	1	1		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXAL	FEAT_LSE
10	0	1	1		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINAL	FEAT_LSE
10	0	1	1		1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPAL	FEAT_LSE
11	0	0	0		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADD	FEAT_LSE
11	0	0	0		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLR	FEAT_LSE
11	0	0	0		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEOR	FEAT_LSE
11	0	0	0		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSET	FEAT_LSE
11	0	0	0		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX	FEAT_LSE
11	0	0	0		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN	FEAT_LSE
11	0	0	0		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX	FEAT_LSE
11	0	0	0		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN	FEAT_LSE
11	0	0	0		1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWP	FEAT_LSE
11	0	0	0		1	010	ST64BV0	FEAT_LS64_V
11	0	0	0		1	011	ST64BV	FEAT_LS64_V
11	0	0	0	11111	1	001	ST64B	FEAT_LS64
11	0	0	0	11111	1	101	LD64B	FEAT_LS64

size	Decode fields				Instruction Details			Feature
	V	A	R	Rs	o3	opc		
11	0	0	1		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDL	FEAT_LSE
11	0	0	1		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRL	FEAT_LSE
11	0	0	1		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORL	FEAT_LSE
11	0	0	1		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETL	FEAT_LSE
11	0	0	1		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXL	FEAT_LSE
11	0	0	1		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINL	FEAT_LSE
11	0	0	1		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXL	FEAT_LSE
11	0	0	1		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINL	FEAT_LSE
11	0	0	1		1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPL	FEAT_LSE
11	0	1	0		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDA	FEAT_LSE
11	0	1	0		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRA	FEAT_LSE
11	0	1	0		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORA	FEAT_LSE
11	0	1	0		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETA	FEAT_LSE
11	0	1	0		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXA	FEAT_LSE
11	0	1	0		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINA	FEAT_LSE
11	0	1	0		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXA	FEAT_LSE
11	0	1	0		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINA	FEAT_LSE
11	0	1	0		1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPA	FEAT_LSE
11	0	1	0		1	100	LDAPR — 64-bit	FEAT_LRCPC
11	0	1	1		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDAL	FEAT_LSE
11	0	1	1		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRAL	FEAT_LSE
11	0	1	1		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORAL	FEAT_LSE
11	0	1	1		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETAL	FEAT_LSE
11	0	1	1		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXAL	FEAT_LSE
11	0	1	1		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINAL	FEAT_LSE
11	0	1	1		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXAL	FEAT_LSE
11	0	1	1		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINAL	FEAT_LSE
11	0	1	1		1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPAL	FEAT_LSE

Load/store register (register offset)

These instructions are under [Loads and Stores](#).

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

Decode fields				Instruction Details
size	V	opc	option	
x1	1	1x		UNALLOCATED
00	0	00	!= 011	STRB (register) — extended register
00	0	00	011	STRB (register) — shifted register
00	0	01	!= 011	LDRB (register) — extended register
00	0	01	011	LDRB (register) — shifted register
00	0	10	!= 011	LDRSB (register) — 64-bit with extended register offset
00	0	10	011	LDRSB (register) — 64-bit with shifted register offset
00	0	11	!= 011	LDRSB (register) — 32-bit with extended register offset
00	0	11	011	LDRSB (register) — 32-bit with shifted register offset
00	1	00	!= 011	STR (register, SIMD&FP)
00	1	00	011	STR (register, SIMD&FP)
00	1	01	!= 011	LDR (register, SIMD&FP)
00	1	01	011	LDR (register, SIMD&FP)
00	1	10		STR (register, SIMD&FP)
00	1	11		LDR (register, SIMD&FP)
01	0	00		STRH (register)
01	0	01		LDRH (register)
01	0	10		LDRSH (register) — 64-bit
01	0	11		LDRSH (register) — 32-bit
01	1	00		STR (register, SIMD&FP)
01	1	01		LDR (register, SIMD&FP)
1x	0	11		UNALLOCATED
1x	1	1x		UNALLOCATED
10	0	00		STR (register) — 32-bit
10	0	01		LDR (register) — 32-bit
10	0	10		LDRSW (register)
10	1	00		STR (register, SIMD&FP)
10	1	01		LDR (register, SIMD&FP)
11	0	00		STR (register) — 64-bit
11	0	01		LDR (register) — 64-bit
11	0	10		PRFM (register)
11	1	00		STR (register, SIMD&FP)
11	1	01		LDR (register, SIMD&FP)

Load/store register (pac)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	V	0	0	M	S	1						imm9					W	1								Rn		Rt

Decode fields				Instruction Details	Feature
size	V	M	W		
!= 11				UNALLOCATED	-
11	0	0	0	LDRAA, LDRAB — key A, offset	FEAT_PAAuth
11	0	0	1	LDRAA, LDRAB — key A, pre-indexed	FEAT_PAAuth
11	0	1	0	LDRAA, LDRAB — key B, offset	FEAT_PAAuth

Decode fields				Instruction Details	Feature
size	V	M	W		
11	0	1	1	LDRAA, LDRAB — key B, pre-indexed	FEAT_PAuth
11	1			UNALLOCATED	-

Load/store register (unsigned immediate)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	V	0	1	opc	imm12														Rn				Rt					

Decode fields			Instruction Details
size	V	opc	
x1	1	1x	UNALLOCATED
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit
00	1	10	STR (immediate, SIMD&FP) — 128-bit
00	1	11	LDR (immediate, SIMD&FP) — 128-bit
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) — 64-bit
01	0	11	LDRSH (immediate) — 32-bit
01	1	00	STR (immediate, SIMD&FP) — 16-bit
01	1	01	LDR (immediate, SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STR (immediate) — 32-bit
10	0	01	LDR (immediate) — 32-bit
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) — 32-bit
10	1	01	LDR (immediate, SIMD&FP) — 32-bit
11	0	00	STR (immediate) — 64-bit
11	0	01	LDR (immediate) — 64-bit
11	0	10	PRFM (immediate)
11	1	00	STR (immediate, SIMD&FP) — 64-bit
11	1	01	LDR (immediate, SIMD&FP) — 64-bit

Data Processing -- Register

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0		op1		101		op2		op3																							

Decode fields				Instruction details
op0	op1	op2	op3	
0	1	0110		Data-processing (2 source)

1	1	0110		Data-processing (1 source)
	0	0xxx		Logical (shifted register)
	0	1xx0		Add/subtract (shifted register)
	0	1xx1		Add/subtract (extended register)
	1	0000	000000	Add/subtract (with carry)
	1	0000	x00001	Rotate right into flags
	1	0000	xx0010	Evaluate into flags
	1	0010	xxxx0x	Conditional compare (register)
	1	0010	xxxx1x	Conditional compare (immediate)
	1	0100		Conditional select
	1	1xxx		Data-processing (3 source)

Data-processing (2 source)

These instructions are under [Data Processing -- Register](#).

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

Decode fields			Instruction Details		Feature
sf	S	opcode			
		000001	UNALLOCATED		-
		011xxx	UNALLOCATED		-
		1xxxxxx	UNALLOCATED		-
	0	00011x	UNALLOCATED		-
	0	001101	UNALLOCATED		-
	0	00111x	UNALLOCATED		-
	1	00001x	UNALLOCATED		-
	1	0001xx	UNALLOCATED		-
	1	001xxx	UNALLOCATED		-
	1	01xxxx	UNALLOCATED		-
0		000000	UNALLOCATED		-
0	0	000010	UDIV — 32-bit		-
0	0	000011	SDIV — 32-bit		-
0	0	00010x	UNALLOCATED		-
0	0	001000	LSLV — 32-bit		-
0	0	001001	LSRV — 32-bit		-
0	0	001010	ASRV — 32-bit		-
0	0	001011	RORV — 32-bit		-
0	0	001100	UNALLOCATED		-
0	0	010x11	UNALLOCATED		-
0	0	010000	CRC32B, CRC32H, CRC32W, CRC32X — CRC32B		-
0	0	010001	CRC32B, CRC32H, CRC32W, CRC32X — CRC32H		-
0	0	010010	CRC32B, CRC32H, CRC32W, CRC32X — CRC32W		-
0	0	010100	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB		-
0	0	010101	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH		-
0	0	010110	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW		-
1	0	000000	SUBP		FEAT_MTE
1	0	000010	UDIV — 64-bit		-
1	0	000011	SDIV — 64-bit		-

Decode fields			Instruction Details	Feature
sf	S	opcode		
1	0	000100	IRG	FEAT_MTE
1	0	000101	GMI	FEAT_MTE
1	0	001000	LSLV — 64-bit	-
1	0	001001	LSRV — 64-bit	-
1	0	001010	ASRV — 64-bit	-
1	0	001011	RORV — 64-bit	-
1	0	001100	PACGA	FEAT_PAuth
1	0	010xx0	UNALLOCATED	-
1	0	010x0x	UNALLOCATED	-
1	0	010011	CRC32B, CRC32H, CRC32W, CRC32X — CRC32X	-
1	0	010111	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX	-
1	1	000000	SUBPS	FEAT_MTE

Data-processing (1 source)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		1	S	1	1	0	1	0	1	1	0	opcode2				opcode				Rn				Rd							

Decode fields				Rn	Instruction Details	Feature
sf	S	opcode2	opcode			
			1xxxxx		UNALLOCATED	-
		xxx1x			UNALLOCATED	-
		xx1xx			UNALLOCATED	-
		x1xxx			UNALLOCATED	-
		1xxxx			UNALLOCATED	-
	0	00000	00011x		UNALLOCATED	-
	0	00000	001xxx		UNALLOCATED	-
	0	00000	01xxxx		UNALLOCATED	-
	1				UNALLOCATED	-
0		00001			UNALLOCATED	-
0	0	00000	000000		RBIT — 32-bit	-
0	0	00000	000001		REV16 — 32-bit	-
0	0	00000	000010		REV — 32-bit	-
0	0	00000	000011		UNALLOCATED	-
0	0	00000	000100		CLZ — 32-bit	-
0	0	00000	000101		CLS — 32-bit	-
1	0	00000	000000		RBIT — 64-bit	-
1	0	00000	000001		REV16 — 64-bit	-
1	0	00000	000010		REV32	-
1	0	00000	000011		REV — 64-bit	-
1	0	00000	000100		CLZ — 64-bit	-
1	0	00000	000101		CLS — 64-bit	-
1	0	00001	000000		PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA	FEAT_PAuth
1	0	00001	000001		PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB	FEAT_PAuth
1	0	00001	000010		PACDA, PACDZA — PACDA	FEAT_PAuth

sf	S	Decode fields		Rn	Instruction Details	Feature
		opcode2	opcode			
1	0	00001	000011		PACDB, PACDZB — PACDB	FEAT_PAuth
1	0	00001	000100		AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA	FEAT_PAuth
1	0	00001	000101		AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB	FEAT_PAuth
1	0	00001	000110		AUTDA, AUTDZA — AUTDA	FEAT_PAuth
1	0	00001	000111		AUTDB, AUTDZB — AUTDB	FEAT_PAuth
1	0	00001	001000	11111	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIZA	FEAT_PAuth
1	0	00001	001001	11111	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIZB	FEAT_PAuth
1	0	00001	001010	11111	PACDA, PACDZA — PACDZA	FEAT_PAuth
1	0	00001	001011	11111	PACDB, PACDZB — PACDZB	FEAT_PAuth
1	0	00001	001100	11111	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIZA	FEAT_PAuth
1	0	00001	001101	11111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIZB	FEAT_PAuth
1	0	00001	001110	11111	AUTDA, AUTDZA — AUTDZA	FEAT_PAuth
1	0	00001	001111	11111	AUTDB, AUTDZB — AUTDZB	FEAT_PAuth
1	0	00001	010000	11111	XPACD, XPACI, XPACLRI — XPACI	FEAT_PAuth
1	0	00001	010001	11111	XPACD, XPACI, XPACLRI — XPACD	FEAT_PAuth
1	0	00001	01001x		UNALLOCATED	-
1	0	00001	0101xx		UNALLOCATED	-
1	0	00001	011xxx		UNALLOCATED	-

Logical (shifted register)

These instructions are under [Data Processing -- Register](#).

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

sf	Decode fields			Instruction Details
	opc	N	imm6	
0			1xxxxx	UNALLOCATED
0	00	0		AND (shifted register) — 32-bit
0	00	1		BIC (shifted register) — 32-bit
0	01	0		ORR (shifted register) — 32-bit
0	01	1		ORN (shifted register) — 32-bit
0	10	0		EOR (shifted register) — 32-bit
0	10	1		EON (shifted register) — 32-bit
0	11	0		ANDS (shifted register) — 32-bit
0	11	1		BICS (shifted register) — 32-bit
1	00	0		AND (shifted register) — 64-bit
1	00	1		BIC (shifted register) — 64-bit
1	01	0		ORR (shifted register) — 64-bit
1	01	1		ORN (shifted register) — 64-bit
1	10	0		EOR (shifted register) — 64-bit
1	10	1		EON (shifted register) — 64-bit
1	11	0		ANDS (shifted register) — 64-bit
1	11	1		BICS (shifted register) — 64-bit

Add/subtract (shifted register)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	0	1	0	1	1	shift	0																						
										Rm						imm6						Rn						Rd			

Decode fields					Instruction Details	
sf	op	S	shift	imm6		
			11		UNALLOCATED	
0				1xxxxx	UNALLOCATED	
0	0	0			ADD (shifted register) — 32-bit	
0	0	1			ADDS (shifted register) — 32-bit	
0	1	0			SUB (shifted register) — 32-bit	
0	1	1			SUBS (shifted register) — 32-bit	
1	0	0			ADD (shifted register) — 64-bit	
1	0	1			ADDS (shifted register) — 64-bit	
1	1	0			SUB (shifted register) — 64-bit	
1	1	1			SUBS (shifted register) — 64-bit	

Add/subtract (extended register)

These instructions are under [Data Processing -- Register](#).

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

Decode fields					Instruction Details	
sf	op	S	opt	imm3		
				1x1	UNALLOCATED	
				11x	UNALLOCATED	
			x1		UNALLOCATED	
			1x		UNALLOCATED	
0	0	0	00		ADD (extended register) — 32-bit	
0	0	1	00		ADDS (extended register) — 32-bit	
0	1	0	00		SUB (extended register) — 32-bit	
0	1	1	00		SUBS (extended register) — 32-bit	
1	0	0	00		ADD (extended register) — 64-bit	
1	0	1	00		ADDS (extended register) — 64-bit	
1	1	0	00		SUB (extended register) — 64-bit	
1	1	1	00		SUBS (extended register) — 64-bit	

Add/subtract (with carry)

These instructions are under [Data Processing -- Register](#).

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

Decode fields			Instruction Details
sf	op	S	
0	0	0	ADC — 32-bit
0	0	1	ADCS — 32-bit
0	1	0	SBC — 32-bit
0	1	1	SBCS — 32-bit

Decode fields			Instruction Details
sf	op	S	
1	0	0	ADC — 64-bit
1	0	1	ADCS — 64-bit
1	1	0	SBC — 64-bit
1	1	1	SBCS — 64-bit

Rotate right into flags

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	0	0	imm6				0	0	0	0	1	Rn				o2	mask						

Decode fields				Instruction Details	Feature
sf	op	S	o2		
0				UNALLOCATED	-
1	0	0		UNALLOCATED	-
1	0	1	0	RMIF	FEAT_FlagM
1	0	1	1	UNALLOCATED	-
1	1			UNALLOCATED	-

Evaluate into flags

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
sf		op		S		1		1		0		1		0		0		0		0		opcode2						sz		0		0		1		0		Rn				o3		mask			

sf	op	S	Decode fields				Instruction Details	Feature
			opcode2	sz	o3	mask		
0	0	0					UNALLOCATED	-
0	0	1	!= 000000				UNALLOCATED	-
0	0	1	000000		0	!= 1101	UNALLOCATED	-
0	0	1	000000		1		UNALLOCATED	-
0	0	1	000000	0	0	1101	SETF8, SETF16 — SETF8	FEAT_FlagM
0	0	1	000000	1	0	1101	SETF8, SETF16 — SETF16	FEAT_FlagM
0	1						UNALLOCATED	-
1							UNALLOCATED	-

Conditional compare (register)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	1	0	Rm				cond				0	o2	Rn				o3	nzcv					

Decode fields					Instruction Details
sf	op	S	o2	o3	
				1	UNALLOCATED
			1		UNALLOCATED
		0			UNALLOCATED
0	0	1	0	0	CCMN (register) — 32-bit
0	1	1	0	0	CCMP (register) — 32-bit

Decode fields					Instruction Details
sf	op	S	o2	o3	
1	0	1	0	0	CCMN (register) — 64-bit
1	1	1	0	0	CCMP (register) — 64-bit

Conditional compare (immediate)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	1	0	imm5					cond					1	o2	Rn					o3	nzcw		

Decode fields					Instruction Details
sf	op	S	o2	o3	
				1	UNALLOCATED
			1		UNALLOCATED
		0			UNALLOCATED
0	0	1	0	0	CCMN (immediate) — 32-bit
0	1	1	0	0	CCMP (immediate) — 32-bit
1	0	1	0	0	CCMN (immediate) — 64-bit
1	1	1	0	0	CCMP (immediate) — 64-bit

Conditional select

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	1	0	0	Rm					cond					op2	Rn					Rd				

Decode fields				Instruction Details
sf	op	S	op2	
			1x	UNALLOCATED
		1		UNALLOCATED
0	0	0	00	CSEL — 32-bit
0	0	0	01	CSINC — 32-bit
0	1	0	00	CSINV — 32-bit
0	1	0	01	CSNEG — 32-bit
1	0	0	00	CSEL — 64-bit
1	0	0	01	CSINC — 64-bit
1	1	0	00	CSINV — 64-bit
1	1	0	01	CSNEG — 64-bit

Data-processing (3 source)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op54		1	1	0	1	1	op31				Rm					o0	Ra					Rn					Rd			

Decode fields				Instruction Details
sf	op54	op31	o0	
	00	010	1	UNALLOCATED
	00	011		UNALLOCATED
	00	100		UNALLOCATED

sf	Decode fields		o0	Instruction Details
	op54	op31		
	00	110	1	UNALLOCATED
	00	111		UNALLOCATED
	01			UNALLOCATED
	1x			UNALLOCATED
0	00	000	0	MADD — 32-bit
0	00	000	1	MSUB — 32-bit
0	00	001	0	UNALLOCATED
0	00	001	1	UNALLOCATED
0	00	010	0	UNALLOCATED
0	00	101	0	UNALLOCATED
0	00	101	1	UNALLOCATED
0	00	110	0	UNALLOCATED
1	00	000	0	MADD — 64-bit
1	00	000	1	MSUB — 64-bit
1	00	001	0	SMADDL
1	00	001	1	SMSUBL
1	00	010	0	SMULH
1	00	101	0	UMADDL
1	00	101	1	UMSUBL
1	00	110	0	UMULH

Data Processing -- Scalar Floating-Point and Advanced SIMD

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0				111			op1		op2			op3																			

Decode fields				Instruction details	Architecture version
op0	op1	op2	op3		
0000	0x	x101	00xxxxx10	UNALLOCATED	-
0010	0x	x101	00xxxxx10	UNALLOCATED	-
0100	0x	x101	00xxxxx10	Cryptographic AES	-
0101	0x	x0xx	xxx0xxx00	Cryptographic three-register SHA	-
0101	0x	x0xx	xxx0xxx10	UNALLOCATED	-
0101	0x	x101	00xxxxx10	Cryptographic two-register SHA	-
0110	0x	x101	00xxxxx10	UNALLOCATED	-
0111	0x	x0xx	xxx0xxxxx0	UNALLOCATED	-
0111	0x	x101	00xxxxx10	UNALLOCATED	-
01x1	00	00xx	xxx0xxxxx1	Advanced SIMD scalar copy	-
01x1	01	00xx	xxx0xxxxx1	UNALLOCATED	-
01x1	0x	0111	00xxxxx10	UNALLOCATED	-
01x1	0x	10xx	xxx00xxx1	Advanced SIMD scalar three same FP16	-
01x1	0x	10xx	xxx01xxx1	UNALLOCATED	-
01x1	0x	1111	00xxxxx10	Advanced SIMD scalar two-register miscellaneous FP16	-
01x1	0x	x0xx	xxx1xxxxx0	UNALLOCATED	-
01x1	0x	x0xx	xxx1xxxxx1	Advanced SIMD scalar three same extra	-

01x1	0x	x100	00xxxxx10	Advanced SIMD scalar two-register miscellaneous	-
01x1	0x	x110	00xxxxx10	Advanced SIMD scalar pairwise	-
01x1	0x	x1xx	1xxxxxx10	UNALLOCATED	-
01x1	0x	x1xx	x1xxxxx10	UNALLOCATED	-
01x1	0x	x1xx	xxxxxxx00	Advanced SIMD scalar three different	-
01x1	0x	x1xx	xxxxxxx1	Advanced SIMD scalar three same	-
01x1	10		xxxxxxx1	Advanced SIMD scalar shift by immediate	-
01x1	11		xxxxxxx1	UNALLOCATED	-
01x1	1x		xxxxxxx0	Advanced SIMD scalar x indexed element	-
0x00	0x	x0xx	xxx0xxx00	Advanced SIMD table lookup	-
0x00	0x	x0xx	xxx0xxx10	Advanced SIMD permute	-
0x10	0x	x0xx	xxx0xxx0	Advanced SIMD extract	-
0xx0	00	00xx	xxx0xxx1	Advanced SIMD copy	-
0xx0	01	00xx	xxx0xxx1	UNALLOCATED	-
0xx0	0x	0111	00xxxxx10	UNALLOCATED	-
0xx0	0x	10xx	xxx00xxx1	Advanced SIMD three same (FP16)	-
0xx0	0x	10xx	xxx01xxx1	UNALLOCATED	-
0xx0	0x	1111	00xxxxx10	Advanced SIMD two-register miscellaneous (FP16)	-
0xx0	0x	x0xx	xxx1xxx0	UNALLOCATED	-
0xx0	0x	x0xx	xxx1xxx1	Advanced SIMD three-register extension	-
0xx0	0x	x100	00xxxxx10	Advanced SIMD two-register miscellaneous	-
0xx0	0x	x110	00xxxxx10	Advanced SIMD across lanes	-
0xx0	0x	x1xx	1xxxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	x1xxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	xxxxxxx00	Advanced SIMD three different	-
0xx0	0x	x1xx	xxxxxxx1	Advanced SIMD three same	-
0xx0	10	0000	xxxxxxx1	Advanced SIMD modified immediate	-
0xx0	10	!= 0000	xxxxxxx1	Advanced SIMD shift by immediate	-
0xx0	11		xxxxxxx1	UNALLOCATED	-
0xx0	1x		xxxxxxx0	Advanced SIMD vector x indexed element	-
1100	00	10xx	xxx10xxxx	Cryptographic three-register, imm2	-
1100	00	11xx	xxx1x00xx	Cryptographic three-register SHA 512	-
1100	00		xxx0xxxxx	Cryptographic four-register	-
1100	01	00xx		XAR	FEAT_SHA3
1100	01	1000	0001000xx	Cryptographic two-register SHA 512	-
1xx0	1x			UNALLOCATED	-
x0x1	0x	x0xx		Conversion between floating-point and fixed-point	-
x0x1	0x	x1xx	xxx000000	Conversion between floating-point and integer	-
x0x1	0x	x1xx	xxx10000	Floating-point data-processing (1 source)	-
x0x1	0x	x1xx	xxxxx1000	Floating-point compare	-
x0x1	0x	x1xx	xxxxxx100	Floating-point immediate	-
x0x1	0x	x1xx	xxxxxxx01	Floating-point conditional compare	-
x0x1	0x	x1xx	xxxxxxx10	Floating-point data-processing (2 source)	-
x0x1	0x	x1xx	xxxxxxx11	Floating-point conditional select	-
x0x1	1x			Floating-point data-processing (3 source)	-

Cryptographic AES

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields size	opcode	Instruction Details
	x1xxx	UNALLOCATED
	000xx	UNALLOCATED
	1xxxx	UNALLOCATED
x1		UNALLOCATED
00	00100	AESE
00	00101	AESD
00	00110	AESMC
00	00111	AESIMC
1x		UNALLOCATED

Cryptographic three-register SHA

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields size	opcode	Instruction Details
	111	UNALLOCATED
x1		UNALLOCATED
00	000	SHA1C
00	001	SHA1P
00	010	SHA1M
00	011	SHA1SU0
00	100	SHA256H
00	101	SHA256H2
00	110	SHA256SU1
1x		UNALLOCATED

Cryptographic two-register SHA

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields size	opcode	Instruction Details
	xx1xx	UNALLOCATED
	x1xxx	UNALLOCATED
	1xxxx	UNALLOCATED
x1		UNALLOCATED
00	00000	SHA1H
00	00001	SHA1SU1
00	00010	SHA256SU0

Decode fields size	opcode	Instruction Details
00	00011	UNALLOCATED
1x		UNALLOCATED

Advanced SIMD scalar copy

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	op	1	1	1	1	0	0	0	0	imm5				0	imm4				1	Rn				Rd						

Decode fields op	imm4	Instruction Details
0	xxx1	UNALLOCATED
0	xx1x	UNALLOCATED
0	x1xx	UNALLOCATED
0	0000	DUP (element)
0	1xxx	UNALLOCATED
1		UNALLOCATED

Advanced SIMD scalar three same FP16

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields U	a	opcode	Instruction Details	Feature
		110	UNALLOCATED	-
	1	011	UNALLOCATED	-
0	0	011	FMULX	FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_FP16
0	0	101	UNALLOCATED	-
0	0	111	FRECPS	FEAT_FP16
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	111	FRSQRTS	FEAT_FP16
1	0	011	UNALLOCATED	-
1	0	100	FCMGE (register)	FEAT_FP16
1	0	101	FACGE	FEAT_FP16
1	0	111	UNALLOCATED	-
1	1	010	FABD	FEAT_FP16
1	1	100	FCMGT (register)	FEAT_FP16
1	1	101	FACGT	FEAT_FP16
1	1	111	UNALLOCATED	-

Advanced SIMD scalar two-register miscellaneous FP16

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields			Instruction Details	Feature
U	a	opcode		
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	01111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11010	FCVTNS (vector)	FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_FP16
0	1	01100	FCMGT (zero)	FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_FP16
0	1	11010	FCVTPS (vector)	FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_FP16
0	1	11101	FRECPE	FEAT_FP16
0	1	11111	FRECPX	FEAT_FP16
1	0	11010	FCVTNU (vector)	FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_FP16
1	1	01100	FCMGE (zero)	FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_FP16
1	1	01110	UNALLOCATED	-
1	1	11010	FCVTPU (vector)	FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_FP16
1	1	11101	FRSQRT	FEAT_FP16
1	1	11111	UNALLOCATED	-

Advanced SIMD scalar three same extra

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields		Instruction Details	Feature
U	opcode		
	001x	UNALLOCATED	-
	01xx	UNALLOCATED	-
	1xxx	UNALLOCATED	-
0	0000	UNALLOCATED	-
0	0001	UNALLOCATED	-
1	0000	SQRDMLAH (vector)	FEAT_RDM
1	0001	SQRDMLSH (vector)	FEAT_RDM

Advanced SIMD scalar two-register miscellaneous

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields			Instruction Details
U	size	opcode	
		0000x	UNALLOCATED
		00010	UNALLOCATED
		0010x	UNALLOCATED
		00110	UNALLOCATED
		01111	UNALLOCATED
		1000x	UNALLOCATED
		10011	UNALLOCATED
		10101	UNALLOCATED
		10111	UNALLOCATED
		1100x	UNALLOCATED
		11110	UNALLOCATED
	0x	011xx	UNALLOCATED
	0x	11111	UNALLOCATED
	1x	10110	UNALLOCATED
	1x	11100	UNALLOCATED
0		00011	SUQADD
0		00111	SQABS
0		01000	CMGT (zero)
0		01001	CMEQ (zero)
0		01010	CMLT (zero)
0		01011	ABS
0		10010	UNALLOCATED
0		10100	SQXTN, SQXTN2
0	0x	10110	UNALLOCATED
0	0x	11010	FCVTNS (vector)
0	0x	11011	FCVTMS (vector)
0	0x	11100	FCVTAS (vector)
0	0x	11101	SCVTF (vector, integer)
0	1x	01100	FCMGT (zero)
0	1x	01101	FCMEQ (zero)
0	1x	01110	FCMLT (zero)
0	1x	11010	FCVTPS (vector)
0	1x	11011	FCVTZS (vector, integer)
0	1x	11101	FRECPE
0	1x	11111	FRECPX
1		00011	USQADD
1		00111	SQNEG
1		01000	CMGE (zero)
1		01001	CMLE (zero)
1		01010	UNALLOCATED
1		01011	NEG (vector)
1		10010	SQXTUN, SQXTUN2

Decode fields			Instruction Details
U	size	opcode	
1		10100	UQXTN, UQXTN2
1	0x	10110	FCVTXN, FCVTXN2
1	0x	11010	FCVTNU (vector)
1	0x	11011	FCVTMU (vector)
1	0x	11100	FCVTAU (vector)
1	0x	11101	UCVTF (vector, integer)
1	1x	01100	FCMGE (zero)
1	1x	01101	FCMLE (zero)
1	1x	01110	UNALLOCATED
1	1x	11010	FCVTPU (vector)
1	1x	11011	FCVTZU (vector, integer)
1	1x	11101	FRSQRT
1	1x	11111	UNALLOCATED

Advanced SIMD scalar pairwise

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields			Instruction Details	Feature
U	size	opcode		
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11010	UNALLOCATED	-
		111xx	UNALLOCATED	-
	1x	01101	UNALLOCATED	-
0		11011	ADDP (scalar)	-
0	0x	01100	FMAXNMP (scalar) — half-precision	FEAT_FP16
0	0x	01101	FADDP (scalar) — half-precision	FEAT_FP16
0	0x	01111	FMAXP (scalar) — half-precision	FEAT_FP16
0	1x	01100	FMINNMP (scalar) — half-precision	FEAT_FP16
0	1x	01111	FMINP (scalar) — half-precision	FEAT_FP16
1		11011	UNALLOCATED	-
1	0x	01100	FMAXNMP (scalar) — single-precision and double-precision	-
1	0x	01101	FADDP (scalar) — single-precision and double-precision	-
1	0x	01111	FMAXP (scalar) — single-precision and double-precision	-
1	1x	01100	FMINNMP (scalar) — single-precision and double-precision	-
1	1x	01111	FMINP (scalar) — single-precision and double-precision	-

Advanced SIMD scalar three different

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields		Instruction Details
U	opcode	
	00xx	UNALLOCATED
	01xx	UNALLOCATED
	1000	UNALLOCATED
	1010	UNALLOCATED
	1100	UNALLOCATED
	111x	UNALLOCATED
0	1001	SQDMLAL, SQDMLAL2 (vector)
0	1011	SQDMLSL, SQDMLSL2 (vector)
0	1101	SQDMULL, SQDMULL2 (vector)
1	1001	UNALLOCATED
1	1011	UNALLOCATED
1	1101	UNALLOCATED

Advanced SIMD scalar three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields		Instruction Details
U	size opcode	
	00000	UNALLOCATED
	0001x	UNALLOCATED
	00100	UNALLOCATED
	011xx	UNALLOCATED
	1001x	UNALLOCATED
	1x 11011	UNALLOCATED
0	00001	SQADD
0	00101	SQSUB
0	00110	CMGT (register)
0	00111	CMGE (register)
0	01000	SSHL
0	01001	SQSHL (register)
0	01010	SRSHL
0	01011	SQRSHL
0	10000	ADD (vector)
0	10001	CMTST
0	10100	UNALLOCATED
0	10101	UNALLOCATED
0	10110	SQDMULH (vector)
0	10111	UNALLOCATED
0	0x 11000	UNALLOCATED
0	0x 11001	UNALLOCATED
0	0x 11010	UNALLOCATED
0	0x 11011	FMULX
0	0x 11100	FCMEQ (register)
0	0x 11101	UNALLOCATED
0	0x 11110	UNALLOCATED

U	Decode fields		Instruction Details
	size	opcode	
0	0x	11111	FRECPS
0	1x	11000	UNALLOCATED
0	1x	11001	UNALLOCATED
0	1x	11010	UNALLOCATED
0	1x	11100	UNALLOCATED
0	1x	11101	UNALLOCATED
0	1x	11110	UNALLOCATED
0	1x	11111	FRSQRTS
1		00001	UQADD
1		00101	UQSUB
1		00110	CMHI (register)
1		00111	CMHS (register)
1		01000	USHL
1		01001	UQSHL (register)
1		01010	URSHL
1		01011	UQRSHL
1		10000	SUB (vector)
1		10001	CMEQ (register)
1		10100	UNALLOCATED
1		10101	UNALLOCATED
1		10110	SQRDMULH (vector)
1		10111	UNALLOCATED
1	0x	11000	UNALLOCATED
1	0x	11001	UNALLOCATED
1	0x	11010	UNALLOCATED
1	0x	11011	UNALLOCATED
1	0x	11100	FCMGE (register)
1	0x	11101	FACGE
1	0x	11110	UNALLOCATED
1	0x	11111	UNALLOCATED
1	1x	11000	UNALLOCATED
1	1x	11001	UNALLOCATED
1	1x	11010	FABD
1	1x	11100	FCMGT (register)
1	1x	11101	FACGT
1	1x	11110	UNALLOCATED
1	1x	11111	UNALLOCATED

Advanced SIMD scalar shift by immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

U	Decode fields		Instruction Details
	immh	opcode	
	!= 0000	00001	UNALLOCATED
	!= 0000	00011	UNALLOCATED

U	Decode fields		Instruction Details
	immh	opcode	
	!= 0000	00101	UNALLOCATED
	!= 0000	00111	UNALLOCATED
	!= 0000	01001	UNALLOCATED
	!= 0000	01011	UNALLOCATED
	!= 0000	01101	UNALLOCATED
	!= 0000	01111	UNALLOCATED
	!= 0000	101xx	UNALLOCATED
	!= 0000	110xx	UNALLOCATED
	!= 0000	11101	UNALLOCATED
	!= 0000	11110	UNALLOCATED
	0000		UNALLOCATED
0	!= 0000	00000	SSHR
0	!= 0000	00010	SSRA
0	!= 0000	00100	SRRSHR
0	!= 0000	00110	SRRSRA
0	!= 0000	01000	UNALLOCATED
0	!= 0000	01010	SHL
0	!= 0000	01100	UNALLOCATED
0	!= 0000	01110	SQSHL (immediate)
0	!= 0000	10000	UNALLOCATED
0	!= 0000	10001	UNALLOCATED
0	!= 0000	10010	SQSHRN, SQSHRN2
0	!= 0000	10011	SQRSHRN, SQRSHRN2
0	!= 0000	11100	SCVTF (vector, fixed-point)
0	!= 0000	11111	FCVTZS (vector, fixed-point)
1	!= 0000	00000	USHR
1	!= 0000	00010	USRA
1	!= 0000	00100	URSHR
1	!= 0000	00110	URSRA
1	!= 0000	01000	SRI
1	!= 0000	01010	SLI
1	!= 0000	01100	SQSHLU
1	!= 0000	01110	UQSHL (immediate)
1	!= 0000	10000	SQSHRUN, SQSHRUN2
1	!= 0000	10001	SQRSHRUN, SQRSHRUN2
1	!= 0000	10010	UQSHRN, UQSHRN2
1	!= 0000	10011	UQRSHRN, UQRSHRN2
1	!= 0000	11100	UCVTF (vector, fixed-point)
1	!= 0000	11111	FCVTZU (vector, fixed-point)

Advanced SIMD scalar x indexed element

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields			Instruction Details	Feature
U	size	opcode		
		0000	UNALLOCATED	-
		0010	UNALLOCATED	-
		0100	UNALLOCATED	-
		0110	UNALLOCATED	-
		1000	UNALLOCATED	-
		1010	UNALLOCATED	-
		1110	UNALLOCATED	-
	01	0001	UNALLOCATED	-
	01	0101	UNALLOCATED	-
	01	1001	UNALLOCATED	-
0		0011	SQDMLAL, SQDMLAL2 (by element)	-
0		0111	SQDMLSL, SQDMLSL2 (by element)	-
0		1011	SQDMULL, SQDMULL2 (by element)	-
0		1100	SQDMULH (by element)	-
0		1101	SQRDMULH (by element)	-
0		1111	UNALLOCATED	-
0	00	0001	FMLA (by element) — half-precision	FEAT_FP16
0	00	0101	FMLS (by element) — half-precision	FEAT_FP16
0	00	1001	FMUL (by element) — half-precision	FEAT_FP16
0	1x	0001	FMLA (by element) — single-precision and double-precision	-
0	1x	0101	FMLS (by element) — single-precision and double-precision	-
0	1x	1001	FMUL (by element) — single-precision and double-precision	-
1		0011	UNALLOCATED	-
1		0111	UNALLOCATED	-
1		1011	UNALLOCATED	-
1		1100	UNALLOCATED	-
1		1101	SQRDMLAH (by element)	FEAT_RDM
1		1111	SQRDMLSH (by element)	FEAT_RDM
1	00	0001	UNALLOCATED	-
1	00	0101	UNALLOCATED	-
1	00	1001	FMULX (by element) — half-precision	FEAT_FP16
1	1x	0001	UNALLOCATED	-
1	1x	0101	UNALLOCATED	-
1	1x	1001	FMULX (by element) — single-precision and double-precision	-

Advanced SIMD table lookup

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	op2	0	Rm				0	len	op	0	0	Rn				Rd								

Decode fields			Instruction Details
op2	len	op	
x1			UNALLOCATED
00	00	0	TBL — single register table
00	00	1	TBX — single register table
00	01	0	TBL — two register table
00	01	1	TBX — two register table

Decode fields			Instruction Details
op2	len	op	
00	10	0	TBL — three register table
00	10	1	TBX — three register table
00	11	0	TBL — four register table
00	11	1	TBX — four register table
1x			UNALLOCATED

Advanced SIMD permute

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields opcode	Instruction Details
000	UNALLOCATED
001	UZP1
010	TRN1
011	ZIP1
100	UNALLOCATED
101	UZP2
110	TRN2
111	ZIP2

Advanced SIMD extract

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	op2	0		Rm	0	imm4	0		Rn															

Decode fields op2	Instruction Details
x1	UNALLOCATED
00	EXT
1x	UNALLOCATED

Advanced SIMD copy

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	0	0	0	0		imm5	0	imm4	1		Rn														

Q	op	Decode fields imm5	imm4	Instruction Details
		x0000		UNALLOCATED
	0		0000	DUP (element)
	0		0001	DUP (general)
	0		0010	UNALLOCATED
	0		0100	UNALLOCATED
	0		0110	UNALLOCATED

Decode fields				Instruction Details
Q	op	imm5	imm4	
	0		1xxx	UNALLOCATED
0	0		0011	UNALLOCATED
0	0		0101	SMOV
0	0		0111	UMOV
0	1			UNALLOCATED
1	0		0011	INS (general)
1	0		0101	SMOV
1	0	x1000	0111	UMOV
1	1			INS (element)

Advanced SIMD three same (FP16)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields			Instruction Details	Feature
U	a	opcode		
0	0	000	FMAXNM (vector)	FEAT_FP16
0	0	001	FMLA (vector)	FEAT_FP16
0	0	010	FADD (vector)	FEAT_FP16
0	0	011	FMULX	FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_FP16
0	0	101	UNALLOCATED	-
0	0	110	FMAX (vector)	FEAT_FP16
0	0	111	FRECPS	FEAT_FP16
0	1	000	FMINNM (vector)	FEAT_FP16
0	1	001	FMLS (vector)	FEAT_FP16
0	1	010	FSUB (vector)	FEAT_FP16
0	1	011	UNALLOCATED	-
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	110	FMIN (vector)	FEAT_FP16
0	1	111	FRSQRTS	FEAT_FP16
1	0	000	FMAXNMP (vector)	FEAT_FP16
1	0	001	UNALLOCATED	-
1	0	010	FADDP (vector)	FEAT_FP16
1	0	011	FMUL (vector)	FEAT_FP16
1	0	100	FCMGE (register)	FEAT_FP16
1	0	101	FACGE	FEAT_FP16
1	0	110	FMAXP (vector)	FEAT_FP16
1	0	111	FDIV (vector)	FEAT_FP16
1	1	000	FMINNMP (vector)	FEAT_FP16
1	1	001	UNALLOCATED	-
1	1	010	FABD	FEAT_FP16
1	1	011	UNALLOCATED	-
1	1	100	FCMGT (register)	FEAT_FP16
1	1	101	FACGT	FEAT_FP16

Decode fields			Instruction Details	Feature
U	a	opcode		
1	1	110	FMINP (vector)	FEAT_FP16
1	1	111	UNALLOCATED	-

Advanced SIMD two-register miscellaneous (FP16)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields			Instruction Details	Feature
U	a	opcode		
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11000	FRINTN (vector)	FEAT_FP16
0	0	11001	FRINTM (vector)	FEAT_FP16
0	0	11010	FCVTNS (vector)	FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_FP16
0	1	01100	FCMGT (zero)	FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_FP16
0	1	01111	FABS (vector)	FEAT_FP16
0	1	11000	FRINTP (vector)	FEAT_FP16
0	1	11001	FRINTZ (vector)	FEAT_FP16
0	1	11010	FCVTPS (vector)	FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_FP16
0	1	11101	FRECPE	FEAT_FP16
0	1	11111	UNALLOCATED	-
1	0	11000	FRINTA (vector)	FEAT_FP16
1	0	11001	FRINTX (vector)	FEAT_FP16
1	0	11010	FCVTNU (vector)	FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_FP16
1	1	01100	FCMGE (zero)	FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_FP16
1	1	01110	UNALLOCATED	-
1	1	01111	FNEG (vector)	FEAT_FP16
1	1	11000	UNALLOCATED	-
1	1	11001	FRINTI (vector)	FEAT_FP16
1	1	11010	FCVTPU (vector)	FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_FP16

Decode fields			Instruction Details	Feature
U	a	opcode		
1	1	11101	FRSQRT	FEAT_FP16
1	1	11111	FSQRT (vector)	FEAT_FP16

Advanced SIMD three-register extension

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields				Instruction Details	Feature
Q	U	size	opcode		
		0x	0011	UNALLOCATED	-
		11	0011	UNALLOCATED	-
	0		0000	UNALLOCATED	-
	0		0001	UNALLOCATED	-
	0		0010	SDOT (vector)	FEAT_DotProd
	0		1xxx	UNALLOCATED	-
	0	10	0011	USDOT (vector)	FEAT_I8MM
	1		0000	SQRDMLAH (vector)	FEAT_RDM
	1		0001	SQRDMLSH (vector)	FEAT_RDM
	1		0010	UDOT (vector)	FEAT_DotProd
	1		10xx	FCMLA	FEAT_FCMA
	1		11x0	FCADD	FEAT_FCMA
	1	00	1101	UNALLOCATED	-
	1	00	1111	UNALLOCATED	-
	1	01	1111	BFDOT (vector)	FEAT_BF16
	1	1x	1101	UNALLOCATED	-
	1	10	0011	UNALLOCATED	-
	1	10	1111	UNALLOCATED	-
	1	11	1111	BFMLALB, BFMLALT (vector)	FEAT_BF16
0			01xx	UNALLOCATED	-
0	1	01	1101	UNALLOCATED	-
1		0x	01xx	UNALLOCATED	-
1		1x	011x	UNALLOCATED	-
1	0	10	0100	SMMLA (vector)	FEAT_I8MM
1	0	10	0101	USMMLA (vector)	FEAT_I8MM
1	1	01	1101	BFMMLA	FEAT_BF16
1	1	10	0100	UMMLA (vector)	FEAT_I8MM
1	1	10	0101	UNALLOCATED	-

Advanced SIMD two-register miscellaneous

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

U	Decode fields size opcode	Instruction Details	Feature
		1000x	UNALLOCATED
		10101	UNALLOCATED
	0x	011xx	UNALLOCATED
	1x	10111	UNALLOCATED
	1x	11110	UNALLOCATED
	11	10110	UNALLOCATED
0		00000	REV64
0		00001	REV16 (vector)
0		00010	SADDLP
0		00011	SUQADD
0		00100	CLS (vector)
0		00101	CNT
0		00110	SADALP
0		00111	SQABS
0		01000	CMGT (zero)
0		01001	CMEQ (zero)
0		01010	CMLT (zero)
0		01011	ABS
0		10010	XTN, XTN2
0		10011	UNALLOCATED
0		10100	SQXTN, SQXTN2
0	0x	10110	FCVTN, FCVTN2
0	0x	10111	FCVTL, FCVTL2
0	0x	11000	FRINTN (vector)
0	0x	11001	FRINTM (vector)
0	0x	11010	FCVTNS (vector)
0	0x	11011	FCVTMS (vector)
0	0x	11100	FCVTAS (vector)
0	0x	11101	SCVTF (vector, integer)
0	0x	11110	FRINT32Z (vector)
0	0x	11111	FRINT64Z (vector)
0	1x	01100	FCMGT (zero)
0	1x	01101	FCMEQ (zero)
0	1x	01110	FCMLT (zero)
0	1x	01111	FABS (vector)
0	1x	11000	FRINTP (vector)
0	1x	11001	FRINTZ (vector)
0	1x	11010	FCVTPS (vector)
0	1x	11011	FCVTZS (vector, integer)
0	1x	11100	URECPE
0	1x	11101	FRECPE
0	1x	11111	UNALLOCATED
0	10	10110	BFCVTN, BFCVTN2
1		00000	REV32 (vector)
1		00001	UNALLOCATED
1		00010	UADDLP
1		00011	USQADD

Decode fields			Instruction Details	Feature
U	size	opcode		
1		00100	CLZ (vector)	-
1		00110	UADALP	-
1		00111	SQNEG	-
1		01000	CMGE (zero)	-
1		01001	CMLE (zero)	-
1		01010	UNALLOCATED	-
1		01011	NEG (vector)	-
1		10010	SQXTUN, SQXTUN2	-
1		10011	SHLL, SHLL2	-
1		10100	UQXTN, UQXTN2	-
1	0x	10110	FCVTXN, FCVTXN2	-
1	0x	10111	UNALLOCATED	-
1	0x	11000	FRINTA (vector)	-
1	0x	11001	FRINTX (vector)	-
1	0x	11010	FCVTNU (vector)	-
1	0x	11011	FCVTMU (vector)	-
1	0x	11100	FCVTAU (vector)	-
1	0x	11101	UCVTF (vector, integer)	-
1	0x	11110	FRINT32X (vector)	FEAT_FRINTTS
1	0x	11111	FRINT64X (vector)	FEAT_FRINTTS
1	00	00101	NOT	-
1	01	00101	RBIT (vector)	-
1	1x	00101	UNALLOCATED	-
1	1x	01100	FCMGE (zero)	-
1	1x	01101	FCMLE (zero)	-
1	1x	01110	UNALLOCATED	-
1	1x	01111	FNEG (vector)	-
1	1x	11000	UNALLOCATED	-
1	1x	11001	FRINTI (vector)	-
1	1x	11010	FCVTPU (vector)	-
1	1x	11011	FCVTZU (vector, integer)	-
1	1x	11100	URSQRTE	-
1	1x	11101	FRSQRTE	-
1	1x	11111	FSQRT (vector)	-
1	10	10110	UNALLOCATED	-

Advanced SIMD across lanes

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields			Instruction Details	Feature
U	size	opcode		
		0000x	UNALLOCATED	-
		00010	UNALLOCATED	-
		001xx	UNALLOCATED	-
		0100x	UNALLOCATED	-

Decode fields			Instruction Details	Feature
U	size	opcode		
		01011	UNALLOCATED	-
		01101	UNALLOCATED	-
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		111xx	UNALLOCATED	-
0		00011	SADDLV	-
0		01010	SMAXV	-
0		11010	SMINV	-
0		11011	ADDV	-
0	00	01100	FMAXNMV — half-precision	FEAT_FP16
0	00	01111	FMAXV — half-precision	FEAT_FP16
0	01	01100	UNALLOCATED	-
0	01	01111	UNALLOCATED	-
0	10	01100	FMINNMV — half-precision	FEAT_FP16
0	10	01111	FMINV — half-precision	FEAT_FP16
0	11	01100	UNALLOCATED	-
0	11	01111	UNALLOCATED	-
1		00011	UADDLV	-
1		01010	UMAXV	-
1		11010	UMINV	-
1		11011	UNALLOCATED	-
1	0x	01100	FMAXNMV — single-precision and double-precision	-
1	0x	01111	FMAXV — single-precision and double-precision	-
1	1x	01100	FMINNMV — single-precision and double-precision	-
1	1x	01111	FMINV — single-precision and double-precision	-

Advanced SIMD three different

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields		Instruction Details
U	opcode	
	1111	UNALLOCATED
0	0000	SADDL, SADDL2
0	0001	SADDW, SADDW2
0	0010	SSUBL, SSUBL2
0	0011	SSUBW, SSUBW2
0	0100	ADDHN, ADDHN2
0	0101	SABAL, SABAL2
0	0110	SUBHN, SUBHN2
0	0111	SABDL, SABDL2
0	1000	SMLAL, SMLAL2 (vector)
0	1001	SQDMLAL, SQDMLAL2 (vector)
0	1010	SMLSL, SMLSL2 (vector)
0	1011	SQDMLSL, SQDMLSL2 (vector)

Decode fields		Instruction Details
U	opcode	
0	1100	SMULL, SMULL2 (vector)
0	1101	SQDMULL, SQDMULL2 (vector)
0	1110	PMULL, PMULL2
1	0000	UADDL, UADDL2
1	0001	UADDW, UADDW2
1	0010	USUBL, USUBL2
1	0011	USUBW, USUBW2
1	0100	RADDHN, RADDHN2
1	0101	UABAL, UABAL2
1	0110	RSUBHN, RSUBHN2
1	0111	UABDL, UABDL2
1	1000	UMLAL, UMLAL2 (vector)
1	1001	UNALLOCATED
1	1010	UMLSL, UMLSL2 (vector)
1	1011	UNALLOCATED
1	1100	UMULL, UMULL2 (vector)
1	1101	UNALLOCATED
1	1110	UNALLOCATED

Advanced SIMD three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields			Instruction Details	Feature
U	size	opcode		
0		00000	SHADD	-
0		00001	SQADD	-
0		00010	SRHADD	-
0		00100	SHSUB	-
0		00101	SQSUB	-
0		00110	CMGT (register)	-
0		00111	CMGE (register)	-
0		01000	SSHL	-
0		01001	SQSHL (register)	-
0		01010	SRSHL	-
0		01011	SQRSHL	-
0		01100	SMAX	-
0		01101	SMIN	-
0		01110	SABD	-
0		01111	SABA	-
0		10000	ADD (vector)	-
0		10001	CMTST	-
0		10010	MLA (vector)	-
0		10011	MUL (vector)	-
0		10100	SMAXP	-
0		10101	SMINP	-

U	Decode fields		Instruction Details	Feature
	size	opcode		
0		10110	SQDMULH (vector)	-
0		10111	ADDP (vector)	-
0	0x	11000	FMAXNM (vector)	-
0	0x	11001	FMLA (vector)	-
0	0x	11010	FADD (vector)	-
0	0x	11011	FMULX	-
0	0x	11100	FCMEQ (register)	-
0	0x	11110	FMAX (vector)	-
0	0x	11111	FRECPS	-
0	00	00011	AND (vector)	-
0	00	11101	FMLAL, FMLAL2 (vector) — FMLAL	FEAT_FHM
0	01	00011	BIC (vector, register)	-
0	01	11101	UNALLOCATED	-
0	1x	11000	FMINNM (vector)	-
0	1x	11001	FMLS (vector)	-
0	1x	11010	FSUB (vector)	-
0	1x	11011	UNALLOCATED	-
0	1x	11100	UNALLOCATED	-
0	1x	11110	FMIN (vector)	-
0	1x	11111	FRSQRTS	-
0	10	00011	ORR (vector, register)	-
0	10	11101	FMLSL, FMLSL2 (vector) — FMLSL	FEAT_FHM
0	11	00011	ORN (vector)	-
0	11	11101	UNALLOCATED	-
1		00000	UHADD	-
1		00001	UQADD	-
1		00010	URHADD	-
1		00100	UHSUB	-
1		00101	UQSUB	-
1		00110	CMHI (register)	-
1		00111	CMHS (register)	-
1		01000	USHL	-
1		01001	UQSHL (register)	-
1		01010	URSHL	-
1		01011	UQRSHL	-
1		01100	UMAX	-
1		01101	UMIN	-
1		01110	UABD	-
1		01111	UABA	-
1		10000	SUB (vector)	-
1		10001	CMEQ (register)	-
1		10010	MLS (vector)	-
1		10011	PMUL	-
1		10100	UMAXP	-
1		10101	UMINP	-
1		10110	SQRDMULH (vector)	-
1		10111	UNALLOCATED	-

U	Decode fields size	opcode	Instruction Details	Feature
1	0x	11000	FMAXNMP (vector)	-
1	0x	11010	FADDP (vector)	-
1	0x	11011	FMUL (vector)	-
1	0x	11100	FCMGE (register)	-
1	0x	11101	FACGE	-
1	0x	11110	FMAXP (vector)	-
1	0x	11111	FDIV (vector)	-
1	00	00011	EOR (vector)	-
1	00	11001	FMLAL, FMLAL2 (vector) — FMLAL2	FEAT_FHM
1	01	00011	BSL	-
1	01	11001	UNALLOCATED	-
1	1x	11000	FMINNMP (vector)	-
1	1x	11010	FABD	-
1	1x	11011	UNALLOCATED	-
1	1x	11100	FCMGT (register)	-
1	1x	11101	FACGT	-
1	1x	11110	FMINP (vector)	-
1	1x	11111	UNALLOCATED	-
1	10	00011	BIT	-
1	10	11001	FMLSL, FMLSL2 (vector) — FMLSL2	FEAT_FHM
1	11	00011	BIF	-
1	11	11001	UNALLOCATED	-

Advanced SIMD modified immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode				o2	1	d	e	f	g	h	Rd				

Q	Decode fields op	cmode	o2	Instruction Details	Feature
	0	0xxx	1	UNALLOCATED	-
	0	0xx0	0	MOVI — 32-bit shifted immediate	-
	0	0xx1	0	ORR (vector, immediate) — 32-bit	-
	0	10xx	1	UNALLOCATED	-
	0	10x0	0	MOVI — 16-bit shifted immediate	-
	0	10x1	0	ORR (vector, immediate) — 16-bit	-
	0	110x	0	MOVI — 32-bit shifting ones	-
	0	110x	1	UNALLOCATED	-
	0	1110	0	MOVI — 8-bit	-
	0	1110	1	UNALLOCATED	-
	0	1111	0	FMOV (vector, immediate) — single-precision	-
	0	1111	1	FMOV (vector, immediate) — half-precision	FEAT_FP16
	1		1	UNALLOCATED	-
	1	0xx0	0	MVNI — 32-bit shifted immediate	-
	1	0xx1	0	BIC (vector, immediate) — 32-bit	-
	1	10x0	0	MVNI — 16-bit shifted immediate	-
	1	10x1	0	BIC (vector, immediate) — 16-bit	-

Decode fields				Instruction Details	Feature
Q	op	cmode	o2		
	1	110x	0	MVNI — 32-bit shifting ones	-
0	1	1110	0	MOVI — 64-bit scalar	-
0	1	1111	0	UNALLOCATED	-
1	1	1110	0	MOVI — 64-bit vector	-
1	1	1111	0	FMOV (vector, immediate) — double-precision	-

Advanced SIMD shift by immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	1	0	!= 0000				immb			opcode				1	Rn				Rd						
immh																															

The following constraints also apply to this encoding: immh != 0000 && immh != 0000

Decode fields		Instruction Details
U	opcode	
	00001	UNALLOCATED
	00011	UNALLOCATED
	00101	UNALLOCATED
	00111	UNALLOCATED
	01001	UNALLOCATED
	01011	UNALLOCATED
	01101	UNALLOCATED
	01111	UNALLOCATED
	10101	UNALLOCATED
	1011x	UNALLOCATED
	110xx	UNALLOCATED
	11101	UNALLOCATED
	11110	UNALLOCATED
0	00000	SSHR
0	00010	SSRA
0	00100	SRSR
0	00110	SRSRA
0	01000	UNALLOCATED
0	01010	SHL
0	01100	UNALLOCATED
0	01110	SQSHL (immediate)
0	10000	SHRN, SHRN2
0	10001	RSHRN, RSHRN2
0	10010	SQSHRN, SQSHRN2
0	10011	SQRSHRN, SQRSHRN2
0	10100	SSHLL, SSHLL2
0	11100	SCVTF (vector, fixed-point)
0	11111	FCVTZS (vector, fixed-point)
1	00000	USHR
1	00010	USRA
1	00100	URSHR

Decode fields		Instruction Details
U	opcode	
1	00110	URSRA
1	01000	SRI
1	01010	SLI
1	01100	SQSHLU
1	01110	UQSHL (immediate)
1	10000	SQSHRUN, SQSHRUN2
1	10001	SQRSHRUN, SQRSHRUN2
1	10010	UQSHRN, UQSHRN2
1	10011	UQRSHRN, UQRSHRN2
1	10100	USHLL, USHLL2
1	11100	UCVTF (vector, fixed-point)
1	11111	FCVTZU (vector, fixed-point)

Advanced SIMD vector x indexed element

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields			Instruction Details	Feature
U	size	opcode		
	01	1001	UNALLOCATED	-
0		0010	SMLAL, SMLAL2 (by element)	-
0		0011	SQDMLAL, SQDMLAL2 (by element)	-
0		0110	SMLSL, SMLSL2 (by element)	-
0		0111	SQDMLSL, SQDMLSL2 (by element)	-
0		1000	MUL (by element)	-
0		1010	SMULL, SMULL2 (by element)	-
0		1011	SQDMULL, SQDMULL2 (by element)	-
0		1100	SQDMULH (by element)	-
0		1101	SQRDMULH (by element)	-
0		1110	SDOT (by element)	FEAT_DotProd
0	0x	0000	UNALLOCATED	-
0	0x	0100	UNALLOCATED	-
0	00	0001	FMLA (by element) — half-precision	FEAT_FP16
0	00	0101	FMLS (by element) — half-precision	FEAT_FP16
0	00	1001	FMUL (by element) — half-precision	FEAT_FP16
0	00	1111	SUDOT (by element)	FEAT_I8MM
0	01	0001	UNALLOCATED	-
0	01	0101	UNALLOCATED	-
0	01	1111	BFDOT (by element)	FEAT_BF16
0	1x	0001	FMLA (by element) — single-precision and double-precision	-
0	1x	0101	FMLS (by element) — single-precision and double-precision	-
0	1x	1001	FMUL (by element) — single-precision and double-precision	-
0	10	0000	FMLAL, FMLAL2 (by element) — FMLAL	FEAT_FHM
0	10	0100	FMLSL, FMLSL2 (by element) — FMLSL	FEAT_FHM
0	10	1111	USDOT (by element)	FEAT_I8MM
0	11	0000	UNALLOCATED	-

Decode fields			Instruction Details	Feature
U	size	opcode		
0	11	0100	UNALLOCATED	-
0	11	1111	BFMLALB, BFMLALT (by element)	FEAT_BF16
1		0000	MLA (by element)	-
1		0010	UMLAL, UMLAL2 (by element)	-
1		0100	MLS (by element)	-
1		0110	UMLSL, UMLSL2 (by element)	-
1		1010	UMULL, UMULL2 (by element)	-
1		1011	UNALLOCATED	-
1		1101	SQRDMLAH (by element)	FEAT_RDM
1		1110	UDOT (by element)	FEAT_DotProd
1		1111	SQRDMLSH (by element)	FEAT_RDM
1	0x	1000	UNALLOCATED	-
1	0x	1100	UNALLOCATED	-
1	00	0001	UNALLOCATED	-
1	00	0011	UNALLOCATED	-
1	00	0101	UNALLOCATED	-
1	00	0111	UNALLOCATED	-
1	00	1001	FMULX (by element) — half-precision	FEAT_FP16
1	01	0xx1	FCMLA (by element)	FEAT_FCMA
1	1x	1001	FMULX (by element) — single-precision and double-precision	-
1	10	0xx1	FCMLA (by element)	FEAT_FCMA
1	10	1000	FMLAL, FMLAL2 (by element) — FMLAL2	FEAT_FHM
1	10	1100	FMLSL, FMLSL2 (by element) — FMLSL2	FEAT_FHM
1	11	0001	UNALLOCATED	-
1	11	0011	UNALLOCATED	-
1	11	0101	UNALLOCATED	-
1	11	0111	UNALLOCATED	-
1	11	1000	UNALLOCATED	-
1	11	1100	UNALLOCATED	-

Cryptographic three-register, imm2

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm					1	0	imm2	opcode	Rn					Rd						

Decode fields opcode	Instruction Details	Feature
00	SM3TT1A	FEAT_SM3
01	SM3TT1B	FEAT_SM3
10	SM3TT2A	FEAT_SM3
11	SM3TT2B	FEAT_SM3

Cryptographic three-register SHA 512

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields O	opcode	Instruction Details	Feature
0	00	SHA512H	FEAT_SHA512
0	01	SHA512H2	FEAT_SHA512
0	10	SHA512SU1	FEAT_SHA512
0	11	RAX1	FEAT_SHA3
1	00	SM3PARTW1	FEAT_SM3
1	01	SM3PARTW2	FEAT_SM3
1	10	SM4EKEY	FEAT_SM4
1	11	UNALLOCATED	-

Cryptographic four-register

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	Op0				Rm			0			Ra					Rn					Rd		

Decode fields Op0	Instruction Details	Feature
00	EOR3	FEAT_SHA3
01	BCAX	FEAT_SHA3
10	SM3SS1	FEAT_SM3
11	UNALLOCATED	-

Cryptographic two-register SHA 512

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields opcode	Instruction Details	Feature
00	SHA512SU0	FEAT_SHA512
01	SM4E	FEAT_SM4
1x	UNALLOCATED	-

Conversion between floating-point and fixed-point

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields						Instruction Details		Feature
sf	S	ptype	rmode	opcode	scale			
				1xx		UNALLOCATED		-
			x0	00x		UNALLOCATED		-
			x1	01x		UNALLOCATED		-
			0x	00x		UNALLOCATED		-
			1x	01x		UNALLOCATED		-
		10				UNALLOCATED		-
	1					UNALLOCATED		-

sf	S	Decode fields			scale	Instruction Details	Feature
		ptype	rmode	opcode			
0					0XXXXX	UNALLOCATED	-
0	0	00	00	010		SCVTF (scalar, fixed-point) — 32-bit to single-precision	-
0	0	00	00	011		UCVTF (scalar, fixed-point) — 32-bit to single-precision	-
0	0	00	11	000		FCVTZS (scalar, fixed-point) — single-precision to 32-bit	-
0	0	00	11	001		FCVTZU (scalar, fixed-point) — single-precision to 32-bit	-
0	0	01	00	010		SCVTF (scalar, fixed-point) — 32-bit to double-precision	-
0	0	01	00	011		UCVTF (scalar, fixed-point) — 32-bit to double-precision	-
0	0	01	11	000		FCVTZS (scalar, fixed-point) — double-precision to 32-bit	-
0	0	01	11	001		FCVTZU (scalar, fixed-point) — double-precision to 32-bit	-
0	0	11	00	010		SCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011		UCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	11	000		FCVTZS (scalar, fixed-point) — half-precision to 32-bit	FEAT_FP16
0	0	11	11	001		FCVTZU (scalar, fixed-point) — half-precision to 32-bit	FEAT_FP16
1	0	00	00	010		SCVTF (scalar, fixed-point) — 64-bit to single-precision	-
1	0	00	00	011		UCVTF (scalar, fixed-point) — 64-bit to single-precision	-
1	0	00	11	000		FCVTZS (scalar, fixed-point) — single-precision to 64-bit	-
1	0	00	11	001		FCVTZU (scalar, fixed-point) — single-precision to 64-bit	-
1	0	01	00	010		SCVTF (scalar, fixed-point) — 64-bit to double-precision	-
1	0	01	00	011		UCVTF (scalar, fixed-point) — 64-bit to double-precision	-
1	0	01	11	000		FCVTZS (scalar, fixed-point) — double-precision to 64-bit	-
1	0	01	11	001		FCVTZU (scalar, fixed-point) — double-precision to 64-bit	-
1	0	11	00	010		SCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011		UCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	11	000		FCVTZS (scalar, fixed-point) — half-precision to 64-bit	FEAT_FP16
1	0	11	11	001		FCVTZU (scalar, fixed-point) — half-precision to 64-bit	FEAT_FP16

Conversion between floating-point and integer

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

sf	S	Decode fields		opcode	Instruction Details	Feature
		ptype	rmode			
			x1	01x	UNALLOCATED	-
			x1	10x	UNALLOCATED	-
			1x	01x	UNALLOCATED	-
			1x	10x	UNALLOCATED	-
	0	10		0xx	UNALLOCATED	-
	0	10		10x	UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	x1	11x	UNALLOCATED	-
0	0	00	00	000	FCVTNS (scalar) — single-precision to 32-bit	-
0	0	00	00	001	FCVTNU (scalar) — single-precision to 32-bit	-
0	0	00	00	010	SCVTF (scalar, integer) — 32-bit to single-precision	-
0	0	00	00	011	UCVTF (scalar, integer) — 32-bit to single-precision	-
0	0	00	00	100	FCVTAS (scalar) — single-precision to 32-bit	-
0	0	00	00	101	FCVTAU (scalar) — single-precision to 32-bit	-
0	0	00	00	110	FMOV (general) — single-precision to 32-bit	-
0	0	00	00	111	FMOV (general) — 32-bit to single-precision	-
0	0	00	01	000	FCVTPS (scalar) — single-precision to 32-bit	-
0	0	00	01	001	FCVTPU (scalar) — single-precision to 32-bit	-
0	0	00	1x	11x	UNALLOCATED	-
0	0	00	10	000	FCVTMS (scalar) — single-precision to 32-bit	-
0	0	00	10	001	FCVTMU (scalar) — single-precision to 32-bit	-
0	0	00	11	000	FCVTZS (scalar, integer) — single-precision to 32-bit	-
0	0	00	11	001	FCVTZU (scalar, integer) — single-precision to 32-bit	-
0	0	01	0x	11x	UNALLOCATED	-
0	0	01	00	000	FCVTNS (scalar) — double-precision to 32-bit	-
0	0	01	00	001	FCVTNU (scalar) — double-precision to 32-bit	-
0	0	01	00	010	SCVTF (scalar, integer) — 32-bit to double-precision	-
0	0	01	00	011	UCVTF (scalar, integer) — 32-bit to double-precision	-
0	0	01	00	100	FCVTAS (scalar) — double-precision to 32-bit	-
0	0	01	00	101	FCVTAU (scalar) — double-precision to 32-bit	-
0	0	01	01	000	FCVTPS (scalar) — double-precision to 32-bit	-
0	0	01	01	001	FCVTPU (scalar) — double-precision to 32-bit	-
0	0	01	10	000	FCVTMS (scalar) — double-precision to 32-bit	-
0	0	01	10	001	FCVTMU (scalar) — double-precision to 32-bit	-
0	0	01	10	11x	UNALLOCATED	-
0	0	01	11	000	FCVTZS (scalar, integer) — double-precision to 32-bit	-
0	0	01	11	001	FCVTZU (scalar, integer) — double-precision to 32-bit	-
0	0	01	11	110	EJCVTZS	FEAT_JSCVT
0	0	01	11	111	UNALLOCATED	-
0	0	10		11x	UNALLOCATED	-
0	0	11	00	000	FCVTNS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	001	FCVTNU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	010	SCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011	UCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	100	FCVTAS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	101	FCVTAU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	110	FMOV (general) — half-precision to 32-bit	FEAT_FP16

sf	S	Decode fields		opcode	Instruction Details	Feature
		ptype	rmode			
0	0	11	00	111	FMOV (general) — 32-bit to half-precision	FEAT_FP16
0	0	11	01	000	FCVTPS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	01	001	FCVTPU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	10	000	FCVTMS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	10	001	FCVTMU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	11	000	FCVTZS (scalar, integer) — half-precision to 32-bit	FEAT_FP16
0	0	11	11	001	FCVTZU (scalar, integer) — half-precision to 32-bit	FEAT_FP16
1	0	00		11x	UNALLOCATED	-
1	0	00	00	000	FCVTNS (scalar) — single-precision to 64-bit	-
1	0	00	00	001	FCVTNU (scalar) — single-precision to 64-bit	-
1	0	00	00	010	SCVTF (scalar, integer) — 64-bit to single-precision	-
1	0	00	00	011	UCVTF (scalar, integer) — 64-bit to single-precision	-
1	0	00	00	100	FCVTAS (scalar) — single-precision to 64-bit	-
1	0	00	00	101	FCVTAU (scalar) — single-precision to 64-bit	-
1	0	00	01	000	FCVTPS (scalar) — single-precision to 64-bit	-
1	0	00	01	001	FCVTPU (scalar) — single-precision to 64-bit	-
1	0	00	10	000	FCVTMS (scalar) — single-precision to 64-bit	-
1	0	00	10	001	FCVTMU (scalar) — single-precision to 64-bit	-
1	0	00	11	000	FCVTZS (scalar, integer) — single-precision to 64-bit	-
1	0	00	11	001	FCVTZU (scalar, integer) — single-precision to 64-bit	-
1	0	01	x1	11x	UNALLOCATED	-
1	0	01	00	000	FCVTNS (scalar) — double-precision to 64-bit	-
1	0	01	00	001	FCVTNU (scalar) — double-precision to 64-bit	-
1	0	01	00	010	SCVTF (scalar, integer) — 64-bit to double-precision	-
1	0	01	00	011	UCVTF (scalar, integer) — 64-bit to double-precision	-
1	0	01	00	100	FCVTAS (scalar) — double-precision to 64-bit	-
1	0	01	00	101	FCVTAU (scalar) — double-precision to 64-bit	-
1	0	01	00	110	FMOV (general) — double-precision to 64-bit	-
1	0	01	00	111	FMOV (general) — 64-bit to double-precision	-
1	0	01	01	000	FCVTPS (scalar) — double-precision to 64-bit	-
1	0	01	01	001	FCVTPU (scalar) — double-precision to 64-bit	-
1	0	01	1x	11x	UNALLOCATED	-
1	0	01	10	000	FCVTMS (scalar) — double-precision to 64-bit	-
1	0	01	10	001	FCVTMU (scalar) — double-precision to 64-bit	-
1	0	01	11	000	FCVTZS (scalar, integer) — double-precision to 64-bit	-
1	0	01	11	001	FCVTZU (scalar, integer) — double-precision to 64-bit	-
1	0	10	x0	11x	UNALLOCATED	-
1	0	10	01	110	FMOV (general) — top half of 128-bit to 64-bit	-
1	0	10	01	111	FMOV (general) — 64-bit to top half of 128-bit	-
1	0	10	1x	11x	UNALLOCATED	-
1	0	11	00	000	FCVTNS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	001	FCVTNU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	010	SCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011	UCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	100	FCVTAS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	101	FCVTAU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	110	FMOV (general) — half-precision to 64-bit	FEAT_FP16

Decode fields					Instruction Details	Feature
sf	S	ptype	rmode	opcode		
1	0	11	00	111	FMOV (general) — 64-bit to half-precision	FEAT_FP16
1	0	11	01	000	FCVTPS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	01	001	FCVTPU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	10	000	FCVTMS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	10	001	FCVTMU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	11	000	FCVTZS (scalar, integer) — half-precision to 64-bit	FEAT_FP16
1	0	11	11	001	FCVTZU (scalar, integer) — half-precision to 64-bit	FEAT_FP16

Floating-point data-processing (1 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields				Instruction Details	Feature
M	S	ptype	opcode		
			1xxxxx	UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	000000	FMOV (register) — single-precision	-
0	0	00	000001	FABS (scalar) — single-precision	-
0	0	00	000010	FNEG (scalar) — single-precision	-
0	0	00	000011	FSQRT (scalar) — single-precision	-
0	0	00	000100	UNALLOCATED	-
0	0	00	000101	FCVT — single-precision to double-precision	-
0	0	00	000110	UNALLOCATED	-
0	0	00	000111	FCVT — single-precision to half-precision	-
0	0	00	001000	FRINTN (scalar) — single-precision	-
0	0	00	001001	FRINTP (scalar) — single-precision	-
0	0	00	001010	FRINTM (scalar) — single-precision	-
0	0	00	001011	FRINTZ (scalar) — single-precision	-
0	0	00	001100	FRINTA (scalar) — single-precision	-
0	0	00	001101	UNALLOCATED	-
0	0	00	001110	FRINTX (scalar) — single-precision	-
0	0	00	001111	FRINTI (scalar) — single-precision	-
0	0	00	010000	FRINT32Z (scalar) — single-precision	FEAT_FRINTTS
0	0	00	010001	FRINT32X (scalar) — single-precision	FEAT_FRINTTS
0	0	00	010010	FRINT64Z (scalar) — single-precision	FEAT_FRINTTS
0	0	00	010011	FRINT64X (scalar) — single-precision	FEAT_FRINTTS
0	0	00	0101xx	UNALLOCATED	-
0	0	00	011xxx	UNALLOCATED	-
0	0	01	000000	FMOV (register) — double-precision	-
0	0	01	000001	FABS (scalar) — double-precision	-
0	0	01	000010	FNEG (scalar) — double-precision	-
0	0	01	000011	FSQRT (scalar) — double-precision	-
0	0	01	000100	FCVT — double-precision to single-precision	-
0	0	01	000101	UNALLOCATED	-
0	0	01	000110	BFCVT	FEAT_BF16
0	0	01	000111	FCVT — double-precision to half-precision	-

Decode fields				Instruction Details	Feature
M	S	ptype	opcode		
0	0	01	001000	FRINTN (scalar) — double-precision	-
0	0	01	001001	FRINTP (scalar) — double-precision	-
0	0	01	001010	FRINTM (scalar) — double-precision	-
0	0	01	001011	FRINTZ (scalar) — double-precision	-
0	0	01	001100	FRINTA (scalar) — double-precision	-
0	0	01	001101	UNALLOCATED	-
0	0	01	001110	FRINTX (scalar) — double-precision	-
0	0	01	001111	FRINTI (scalar) — double-precision	-
0	0	01	010000	FRINT32Z (scalar) — double-precision	FEAT_FRINTTS
0	0	01	010001	FRINT32X (scalar) — double-precision	FEAT_FRINTTS
0	0	01	010010	FRINT64Z (scalar) — double-precision	FEAT_FRINTTS
0	0	01	010011	FRINT64X (scalar) — double-precision	FEAT_FRINTTS
0	0	01	0101xx	UNALLOCATED	-
0	0	01	011xxx	UNALLOCATED	-
0	0	10	0xxxxx	UNALLOCATED	-
0	0	11	000000	FMOV (register) — half-precision	FEAT_FP16
0	0	11	000001	FABS (scalar) — half-precision	FEAT_FP16
0	0	11	000010	FNEG (scalar) — half-precision	FEAT_FP16
0	0	11	000011	FSQRT (scalar) — half-precision	FEAT_FP16
0	0	11	000100	FCVT — half-precision to single-precision	-
0	0	11	000101	FCVT — half-precision to double-precision	-
0	0	11	00011x	UNALLOCATED	-
0	0	11	001000	FRINTN (scalar) — half-precision	FEAT_FP16
0	0	11	001001	FRINTP (scalar) — half-precision	FEAT_FP16
0	0	11	001010	FRINTM (scalar) — half-precision	FEAT_FP16
0	0	11	001011	FRINTZ (scalar) — half-precision	FEAT_FP16
0	0	11	001100	FRINTA (scalar) — half-precision	FEAT_FP16
0	0	11	001101	UNALLOCATED	-
0	0	11	001110	FRINTX (scalar) — half-precision	FEAT_FP16
0	0	11	001111	FRINTI (scalar) — half-precision	FEAT_FP16
0	0	11	01xxxx	UNALLOCATED	-
1				UNALLOCATED	-

Floating-point compare

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	ptype	1	Rm				op	1	0	0	0	Rn				opcode2								

Decode fields				Instruction Details	Feature
M	S	ptype	op		
				xxxx1	UNALLOCATED
				xxx1x	UNALLOCATED
				xx1xx	UNALLOCATED
			x1	UNALLOCATED	-
			1x	UNALLOCATED	-
		10		UNALLOCATED	-
	1			UNALLOCATED	-

M	S	Decode fields		opcode2	Instruction Details	Feature
		p _{type}	op			
0	0	00	00	00000	FCMP	-
0	0	00	00	01000	FCMP	-
0	0	00	00	10000	FCMPE	-
0	0	00	00	11000	FCMPE	-
0	0	01	00	00000	FCMP	-
0	0	01	00	01000	FCMP	-
0	0	01	00	10000	FCMPE	-
0	0	01	00	11000	FCMPE	-
0	0	11	00	00000	FCMP	FEAT_FP16
0	0	11	00	01000	FCMP	FEAT_FP16
0	0	11	00	10000	FCMPE	FEAT_FP16
0	0	11	00	11000	FCMPE	FEAT_FP16
1					UNALLOCATED	-

Floating-point immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	p	type	1	imm8								1	0	0	imm5				Rd					

M	S	Decode fields		Instruction Details	Feature
		p _{type}	imm5		
			xxxx1	UNALLOCATED	-
			xxx1x	UNALLOCATED	-
			xx1xx	UNALLOCATED	-
			x1xxx	UNALLOCATED	-
			1xxxx	UNALLOCATED	-
		10		UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	00000	FMOV (scalar, immediate) — single-precision	-
0	0	01	00000	FMOV (scalar, immediate) — double-precision	-
0	0	11	00000	FMOV (scalar, immediate) — half-precision	FEAT_FP16
1				UNALLOCATED	-

Floating-point conditional compare

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	p	type	1	Rm				cond		0	1	Rn				op	nzc				v			

M	S	Decode fields		Instruction Details	Feature
		p _{type}	op		
		10		UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	0	FCCMP — single-precision	-
0	0	00	1	FCCMPE — single-precision	-
0	0	01	0	FCCMP — double-precision	-
0	0	01	1	FCCMPE — double-precision	-

Decode fields				Instruction Details	Feature
M	S	ptype	op		
0	0	11	0	FCCMP — half-precision	FEAT_FP16
0	0	11	1	FCCMPE — half-precision	FEAT_FP16
1				UNALLOCATED	-

Floating-point data-processing (2 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields				Instruction Details	Feature
M	S	ptype	opcode		
			1xx1	UNALLOCATED	-
			1x1x	UNALLOCATED	-
			11xx	UNALLOCATED	-
		10		UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	0000	FMUL (scalar) — single-precision	-
0	0	00	0001	FDIV (scalar) — single-precision	-
0	0	00	0010	FADD (scalar) — single-precision	-
0	0	00	0011	FSUB (scalar) — single-precision	-
0	0	00	0100	FMAX (scalar) — single-precision	-
0	0	00	0101	FMIN (scalar) — single-precision	-
0	0	00	0110	FMAXNM (scalar) — single-precision	-
0	0	00	0111	FMINNM (scalar) — single-precision	-
0	0	00	1000	FNMUL (scalar) — single-precision	-
0	0	01	0000	FMUL (scalar) — double-precision	-
0	0	01	0001	FDIV (scalar) — double-precision	-
0	0	01	0010	FADD (scalar) — double-precision	-
0	0	01	0011	FSUB (scalar) — double-precision	-
0	0	01	0100	FMAX (scalar) — double-precision	-
0	0	01	0101	FMIN (scalar) — double-precision	-
0	0	01	0110	FMAXNM (scalar) — double-precision	-
0	0	01	0111	FMINNM (scalar) — double-precision	-
0	0	01	1000	FNMUL (scalar) — double-precision	-
0	0	11	0000	FMUL (scalar) — half-precision	FEAT_FP16
0	0	11	0001	FDIV (scalar) — half-precision	FEAT_FP16
0	0	11	0010	FADD (scalar) — half-precision	FEAT_FP16
0	0	11	0011	FSUB (scalar) — half-precision	FEAT_FP16
0	0	11	0100	FMAX (scalar) — half-precision	FEAT_FP16
0	0	11	0101	FMIN (scalar) — half-precision	FEAT_FP16
0	0	11	0110	FMAXNM (scalar) — half-precision	FEAT_FP16
0	0	11	0111	FMINNM (scalar) — half-precision	FEAT_FP16
0	0	11	1000	FNMUL (scalar) — half-precision	FEAT_FP16
1				UNALLOCATED	-

Floating-point conditional select

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

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

Decode fields			Instruction Details		Feature
M	S	ptype			
		10	UNALLOCATED		-
	1		UNALLOCATED		-
0	0	00	FCSEL — single-precision		-
0	0	01	FCSEL — double-precision		-
0	0	11	FCSEL — half-precision		FEAT_FP16
1			UNALLOCATED		-

Floating-point data-processing (3 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	1	ptype	o1	Rm				o0	Ra				Rn				Rd								

Decode fields					Instruction Details	Feature
M	S	ptype	o1	o0		
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	0	0	FMADD — single-precision	-
0	0	00	0	1	FMSUB — single-precision	-
0	0	00	1	0	FNMADD — single-precision	-
0	0	00	1	1	FNMSUB — single-precision	-
0	0	01	0	0	FMADD — double-precision	-
0	0	01	0	1	FMSUB — double-precision	-
0	0	01	1	0	FNMADD — double-precision	-
0	0	01	1	1	FNMSUB — double-precision	-
0	0	11	0	0	FMADD — half-precision	FEAT_FP16
0	0	11	0	1	FMSUB — half-precision	FEAT_FP16
0	0	11	1	0	FNMADD — half-precision	FEAT_FP16
0	0	11	1	1	FNMSUB — half-precision	FEAT_FP16
1					UNALLOCATED	-

Internal version only: isa [v32.21](#)~~v32.15~~, AdvSIMD v29.05, pseudocode [v2021-06_xml](#)~~v2021-03~~, sve [v2021-06_rc2b](#)~~v2021-03_rc2~~; Build timestamp: [2021-06-28T16:41:22](#)~~2021-03-30T21:41:22~~

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages

Pseudocodes

Library pseudocode for aarch32/dc/AArch32.DC

```
// AArch32.DC()
// =====
// Perform Data Cache Operation.

AArch32.DC(bits(32) regval, CacheOp cacheop, CacheOpScope opscope)
    AccType acctype = AccType_DC;
    CacheRecord cache;

    cache.acctype = acctype;
    cache.cacheop = cacheop;
    cache.opscope = opscope;
    cache.cachetype = CacheType_Data;

    if opscope == CacheOpScope_SetWay then
        cache.shareability = Shareability_NSH;
        (cache.set, cache.way, cache.level) = DecodeSW(ZeroExtend(regval), CacheType_Data);

        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            ((!ELUsingAArch32(EL2) && HCR_EL2.SWI0 == '1') || (ELUsingAArch32(EL2) && HCR.SWI0 == '1') ||
            (!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') || (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00'))
            cache.cacheop = CacheOp_CleanInvalidate;
        CACHE_OP(cache);
        return;

    need_translate = DCInstNeedsTranslation(opscope);
    iswrite = cacheop == CacheOp_Invalidate;
    vaddress = regval;

    size = 0; // by default no watchpoint address
    if iswrite then
        size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
        assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
        assert (size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
        vaddress = Align(regval, size);

    cache.translated = need_translate;
    cache.vaddress = ZeroExtend(vaddress);

    if need_translate then
        wasaligned = TRUE;
        memaddrdesc = AArch32.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        memattrs = memaddrdesc.memattrs;
        cache.paddress = memaddrdesc.paddress;
        if opscope == CacheOpScope_PoC then
            cache.shareability = memattrs.shareability;
        else
            cache.shareability = Shareability_NSH;
    else
        cache.shareability = Shareability_UNKNOWN;
        cache.paddress = FullAddress UNKNOWN;

    if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled()
        && ((!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') || (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00'))
        cache.cacheop = CacheOp_CleanInvalidate;

    CACHE_OP(cache);
    return;
```

Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

```
// AArch32.VCRMatch()
// =====

boolean AArch32.VCRMatch(bits(32) vaddress)

if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
    // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
    match_word = Zeros(32);

    if vaddress<31:5> == ExcVectorBase()<31:5> then
        if HaveEL(EL3) && !IsSecure() then
            match_word<UInt(vaddress<4:2>) + 24> = '1';        // Non-secure vectors
        else
            match_word<UInt(vaddress<4:2>) + 0> = '1';        // Secure vectors (or no EL3)

    if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() && vaddress<31:5> == MVBAR<31:5> then
        match_word<UInt(vaddress<4:2>) + 8> = '1';            // Monitor vectors

    // Mask out bits not corresponding to vectors.
    if !HaveEL(EL3) then
        mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
    elsif !ELUsingAArch32(EL3) then
        mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
    else
        mask = '11011110':'00000000':'11011100':'11011110';

    match_word = match_word AND DBGVCR AND mask;
    match = !IsZero(match_word);

    // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
    if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);

    if !IsZero(vaddress<1:0>) && match then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
else
    match = FALSE;

return match;
```

Library pseudocode for aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// =====

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return DBGGEN == HIGH && SPIDEN == HIGH;
```

Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =====
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplementedGetNumBreakpoints();

    enabled = DBGBCR[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);
    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        if !value_mismatch then value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
```



```

// AArch32.BreakpointValueMatch()
// =====
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n, bits(32) vaddress, boolean linked_to)

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n >= NumBreakpointsImplementedGetNumBreakpoints() then
    (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplementedGetNumBreakpoints() - 1, Unpre
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE,FALSE);

// If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
// call from StateMatch for linking).
if DBGBCR[n].E == '0' then return (FALSE,FALSE);

context_aware = (n >= (NumBreakpointsImplementedGetNumBreakpoints() - NumContextAwareBreakpointsImple

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
dbgtype = DBGBCR[n].BT;

if ((dbgtype IN {'011x','11xx'}) && !HaveVirtHostExt() && !HaveV82Debug()) || // Context matching
    (dbgtype == '010x' && HaltOnBreakpointOrWatchpoint()) || // Address mismatch
    (dbgtype != '0x0x' && !context_aware) || // Context matching
    (dbgtype == '1xxx' && !HaveEL(EL2))) then // EL2 extension
    (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE,FALSE);
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (dbgtype == '0x0x');
mismatch   = (dbgtype == '010x');
match_vmid = (dbgtype == '10xx');
match_cid1 = (dbgtype == 'xx1x');
match_cid2 = (dbgtype == '11xx');
linked     = (dbgtype == 'xxx1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, or if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return (FALSE,FALSE);

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return (FALSE,FALSE);

// Do the comparison.
if match_addr then
    byte = UInt(vaddress<1:0>);
    assert byte IN {0,2}; // "vaddress" is halfword aligned
    byte_select_match = (DBGBCR[n].BAS<byte> == '1');
    integer top = 31;
    BVR_match = (vaddress<top:2> == DBGBCR[n]<top:2>) && byte_select_match;

elseif match_cid1 then
    BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBCR[n]<31:0>);
if match_vmid then
    if ELUsingAArch32(EL2) then
        vmid = ZeroExtend(VTTBR.VMID, 16);
        bvr_vmid = ZeroExtend(DBGBCR[n]<7:0>, 16);
    elseif !Have16bitVMID() || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);

```

```

        bvr_vmid = ZeroExtend(DBGXVR[n]<7:0>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGXVR[n]<15:0>;
        BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
            vmid == bvr_vmid);
    elseif match_cid2 then
        BXVR_match = (PSTATE.EL != EL3 && (()) &&
            vmid == bvr_vmid);
    elseif match_cid2 then
        BXVR_match = ((HaveVirtHostExt() || HaveV82Debug()) &&
            EL2Enabled() &&
            !ELUsingAArch32(EL2) &&
            DBGXVR[n]<31:0> == CONTEXTIDR_EL2<31:0>);

    bvr_match_valid = (match_addr || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

    return (match && !mismatch, !match && mismatch);

```

Library pseudocode for aarch32/debug/breakpoint/AArch32.StateMatch

```
// AArch32.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreakpnt, boolean ispriv)
// "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

// If parameters are set to a reserved type, behaves as either disabled or a defined type
(c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpnt);
if c == Constraint_DISABLED then return FALSE;
// Otherwise the HMC,SSC,PxC values are either valid or the values returned by
// CheckValidStateMatch are valid.

PL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
PL1_match = PxC<0> == '1';
PL0_match = PxC<1> == '1';
SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';

if !ispriv && !isbreakpnt then
    priv_match = PL0_match;
elsif SSU_match then
    priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
else
    case PSTATE.EL of
        when EL3 priv_match = PL1_match;           // EL3 and EL1 are both PL1
        when EL2 priv_match = PL2_match;
        when EL1 priv_match = PL1_match;
        when EL0 priv_match = PL0_match;

    case SSC of
        when '00' security_state_match = TRUE;           // Both
        when '01' security_state_match = !IsSecure();    // Non-secure only
        when '10' security_state_match = IsSecure();     // Secure only
        when '11' security_state_match = (HMC == '1' || IsSecure()); // HMC=1 -> Both, 0 -> Secure only

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = NumBreakpointsImplementedGetNumBreakpoints() - NumContextAwareBreakpointsImplemented();
        last_ctx_cmp = NumBreakpointsImplementedGetNumBreakpoints() - 1;
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTX);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED return FALSE;    // Disabled
                when Constraint_NONE linked = FALSE;     // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

        if linked then
            vaddress = bits(32) UNKNOWN;
            linked_to = TRUE;
            (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

```
// AArch32.GenerateDebugExceptions()
// =====

boolean AArch32.GenerateDebugExceptions()
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());
```

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```
// AArch32.GenerateDebugExceptionsFrom()
// =====

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)

    if !ELUsingAArch32(DebugTargetFrom(secure)) then
        mask = '0'; // No PSTATE.D in AArch32 state
        return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    if HaveEL(EL3) && secure then
        assert from != EL2; // Secure EL2 always uses AArch64
        if IsSecureEL2Enabled() then
            // Implies that EL3 and EL2 both using AArch64
            enabled = MDCR_EL3.SDD == '0';
        else
            spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
            if spd<1> == '1' then
                enabled = spd<0> == '1';
            else
                // SPD == 0b01 is reserved, but behaves the same as 0b00.
                enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from != EL2;

    return enabled;
```

Library pseudocode for aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```
// AArch32.CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch32.CheckForPMUOverflow()

    if !ELUsingAArch32(EL1) then return AArch64.CheckForPMUOverflow();
    pmuirq = PMCR.E == '1' && PMINTENSET<31> == '1' && PMOVSSET<31> == '1';
    for n = 0 to NumEventCountersImplementedGetNumEventCounters() - 1
        if HaveEL(EL2) then
            hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
            hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
            E = (if n < UInt(hpmn) then PMCR.E else hpme);
        else
            E = PMCR.E;
        if E == '1' && PMINTENSET<n> == '1' && PMOVSSET<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSLR_EL0.)

    return pmuirq;
```



```

// AArch32.CountEvents()
// =====
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch32.CountEvents(integer n)
    assert n == 31 || n < NumEventCountersImplementedGetNumEventCounters();

    if !ELUsingAArch32(EL1) then return AArch64.CountEvents(n);

    // Event counting is disabled in Debug state
    debug = Halted();

    // In Non-secure state, some counters are reserved for EL2
    if HaveEL(EL2) then
        hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
        hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
        resvd_for_el2 = n >= UInt(hpmn) && n != 31;
    else
        resvd_for_el2 = FALSE;

    // Main enable controls
    if resvd_for_el2 then
        E = if ELUsingAArch32(EL2) then HDCR.HPME else MDCR_EL2.HPME;
    else
        E = PMCR.E;
    enabled = E == '1' && PMCNTENSET<n> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;
    // Event counting in Secure state is prohibited if all of:
    // * EL3 is implemented
    // * One of the following is true:
    //   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
    //     - FEAT_PMuV3p7 is not implemented
    //     - MDCR_EL3.MPMX == 0
    //   - EL3 is using AArch32 and SDCR.SPME == 0
    // * Not executing at EL0, or SDER.SUNIDEN == 0
    if HaveEL(EL3) && IsSecure() then
        spme = if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME;
        if !ELUsingAArch32(EL3) && HavePMuV3p7() then
            prohibited = spme == '0' && MDCR_EL3.MPMX == '0';
        else
            prohibited = spme == '0';
        if prohibited && PSTATE.EL == EL0 then
            prohibited = SDER.SUNIDEN == '0';

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * PMNx is not reserved for EL2
    // * HDCR.HPMD == 1
    if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
        prohibited = HDCR.HPMD == '1';

    // The IMPLEMENTATION DEFINED authentication interface might override software
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // PMCR.DP disables the cycle counter when event counting is prohibited
    if enabled && prohibited && n == 31 then
        enabled = PMCR.DP == '0';

    // If FEAT_PMuV3p5 is implemented, cycle counting can be prohibited.
    // This is not overridden by PMCR.DP.
    if Havev85PMU() && n == 31 then
        if HaveEL(EL3) && IsSecure() then
            sccd = if ELUsingAArch32(EL3) then SDCR.SCCD else MDCR_EL3.SCCD;
            if sccd == '1' then prohibited = TRUE;
        if PSTATE.EL == EL2 && HDCR.HCCD == '1' then
            prohibited = TRUE;

```

```

// Event counting might be frozen
frozen = FALSE;

// If FEAT_PMUV3p7 is implemented, event counting can be frozen
if HavePMUV3p7() && n != 31 then
    ovflw = PM0VSR<NumEventCountersImplementedGetNumEventCounters()-1:0>;
    if resvd_for_el2 then
        FZ = if ELUsingAArch32(EL2) then HDCR.HPMFZ0 else MDCR_EL2.HPMFZ0;
        ovflw<UInt(hpmn)-1:0> = Zeros();
    else
        FZ = PMCR.FZ0;
        if HaveEL(EL2) then
            ovflw<NumEventCountersImplementedGetNumEventCounters()-1:UInt(hpmn)> = Zeros();
        frozen = FZ == '1' && !IsZero(ovflw);

// Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
filter = if n == 31 then PMCCFILTR else PMEVTYPER[n];

P = filter<31>;
U = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';

case PSTATE.EL of
    when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
    when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
    when EL2 filtered = NSH == '0';
    when EL3 filtered = P == '1';

return !debug && enabled && !prohibited && !filtered && !frozen;

```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```

// AArch32.EnterHypModeInDebugState()
// =====
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction();

```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// =====
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    EndOfInstruction();
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// =====
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    EndOfInstruction();
```


Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =====

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)

integer top = 31;
bottom = if DBGWVR[n]<2> == '1' then 2 else 3;           // Word or doubleword
byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
mask = UInt(DBGWCR[n].MASK);

// If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', or
// DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
// UNPREDICTABLE.
if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
    byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
else
    LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1)); MSB = (DBGWCR[n].BAS + LSB);
    if !IsZero(MSB AND (MSB - 1)) then                // Not contiguous
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
        bottom = 3;                                   // For the whole doubleword

// If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask > 0 && mask <= 2 then
    (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
    assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
    case c of
        when Constraint_DISABLED return FALSE;        // Disabled
        when Constraint_NONE     mask = 0;            // No masking
        // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

if mask > bottom then
    // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
    // of DBGxVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
    // included in the match.
    if !IsOnes(DBGxVR_EL1[n]<63:top>) && !IsZero(DBGxVR_EL1[n]<63:top>) then
        if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
            top = 63;
        WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
        WVR_match = vaddress<top:bottom> == DBGWVR[n]<top:bottom>;

return WVR_match && byte_select_match;
```

Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =====
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
                                AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplementedGetNumWatchpoints();

    // "ispriv" is:
    // * FALSE for all loads, stores, and atomic operations executed at EL0.
    // * FALSE if the access is unprivileged.
    // * TRUE for all other loads, stores, and atomic operations.

    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, DBGWCR[n].LBN, isbreakpnt, ispriv);
    ls_match = FALSE;
    ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// =====
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                           (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                           (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress), fault);
    elseif fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(32) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype == Exception_DataAbort;

    exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        exception.ipaddress = ZeroExtend(fault.ipaddress.address);
    else
        exception.ipavalid = FALSE;

    return exception;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// =====

AArch32.CheckPCAlignment()

    bits(32) pc = ThisInstrAddr();
    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

        // Generate an Alignment fault Prefetch Abort exception
        vaddress = pc;
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        secondstage = FALSE;
        AArch32.Abort(vaddress, AlignmentFault(acctype, iswrite, secondstage));
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.// Report syndrome informati

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

    // The encoding used in the IFSR or DFSR can be Long-descriptor format or Short-descriptor
    // format. Normally, the current translation table format determines the format. For an abort
    // from Non-secure state to Monitor mode, the IFSR or DFSR uses the Long-descriptor format if
    // any of the following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * The abort is synchronous and either:
    //   - It is taken from Hyp mode.
    //   - It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = (TTBCR_S.EAE == '1') ||
            (long_format = TTBCR_S.EAE == '1';
        if !IsExternalSyncAbortIsSErrorInterrupt(fault) && ((PSTATE.EL == (fault) && !long_format then
            long_format = PSTATE.EL == EL2 || TTBCR.EAE == '1') ||
                (fault.secondstage && boolean IMPLEMENTATION_DEFINED "Stage 2 synchronous externa
    else
        long_format = TTBCR.EAE == '1';
    d_side = TRUE;
    if long_format then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);
    || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';
    d_side = TRUE;
    if long_format then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType_IC then
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';

    d_side = FALSE;
    if long_format then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// =====// =====

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor =

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (EL2EnabledHaveEL() && PSTATE.EL IN {(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' ||
        (HCR.TGE == '1' || IsSecondStage(fault) ||
        (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
        (IsDebugException(fault) && HDCR.TDE == '1'))));
    bits(32) preferred_exception_return = IsDebugException(fault) && HDCR.TDE == '1' ||
        IsSecondStage(fault));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    if if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress); IsDebugException(fault) then DBGDSCR
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = exception = AArch32.AbortSyndrome( AArch32.AbortSyndrome(Exception_DataAbort,
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);(exception, preferred_exception_return
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// =====// =====

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
route_to_monitor =

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
route_to_hyp = (EL2EnabledHaveEL() && PSTATE.EL IN {(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} &&
(HCR.TGE == '1' ||
(
(HCR.TGE == '1' || IsSecondStage(fault) ||
(HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
(IsDebugException(fault) && HDCR.TDE == '1'))));

bits(32) preferred_exception_return = IsDebugException(fault) && HDCR.TDE == '1' ||
IsSecondStage(fault));

bits(32) preferred_exception_return = ThisInstrAddr();

vect_offset = 0x0C;

lr_offset = 4;

if if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
if route_to_monitor then
AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress); IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
if route_to_monitor then
AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elseif PSTATE.EL == EL2 || route_to_hyp then
if fault.statuscode == Fault_Alignment then // PC Alignment fault
exception = ExceptionSyndrome(Exception_PCAalignment);
exception.vaddress = ThisInstrAddr();
else
exception = exception = AArch32.AbortSyndrome( AArch32.AbortSyndrome(Exception_InstructionAbort));
if PSTATE.EL == EL2 then
AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);(exception, preferred_exception_return, vect_offset);
else
AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// =====

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalFIQException();
    route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR.TGE == '1' || HCR.FMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_FIQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_IRQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```


Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalSErrorException

```
// AArch32.TakePhysicalSErrorException()
// =====

AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) pe_error_state,
                                     bits(25) full_syndrome)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSErrorException(full_syndrome);

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                   (HCR.TGE == '1' || HCR.AMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    bits(2) target_el;
    if route_to_monitor then
        target_el = EL3;
    elseif PSTATE.EL == EL2 || route_to_hyp then
        target_el = EL2;
    else
        target_el = EL1;

    if IsSErrorEdgeTriggered(target_el, full_syndrome) then
        ClearPendingPhysicalSError();

    fault = fault = AsyncExternalAbort(parity, pe_error_state, extflag);
    vaddress = bits(32) UNKNOWN;

    case target_el of
        when AsyncExternalAbort(parity, pe_error_state, extflag);
        vaddress = bits(32) UNKNOWN;

    case target_el of
        when EL3 AArch32.ReportDataAbort(route_to_monitor, fault, vaddress); AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
        when EL2
            exception = exception = AArch32.AbortSyndrome( AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress));
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        when EL1 AArch32.ReportDataAbort(route_to_monitor, fault, vaddress); AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
    otherwise
        Unreachable();
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// =====

AArch32.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then // Virtual IRQ enabled if TGE==0 and FM0==1
        assert HCR.TGE == '0' && HCR.FM0 == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FM0 == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// =====

AArch32.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then // Virtual IRQs enabled if TGE==0 and IM0==1
        assert HCR.TGE == '0' && HCR.IM0 == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IM0 == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualErrorException

```
// AArch32.TakeVirtualErrorException()
// =====

AArch32.TakeVirtualErrorException(bit extflag, bits(2) pe_error_state, bits(25) full_syndrome)

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then // Virtual SError enabled if TGE==0 and AM0==1
        assert HCR.TGE == '0' && HCR.AM0 == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AM0 == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualErrorException(full_syndrome);

    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    if HaveRASExt() then
        if ELUsingAArch32(EL2) then
            fault = AsyncExternalAbort(FALSE, VDFSR.AET, VDFSR.ExT);
        else
            fault = AsyncExternalAbort(FALSE, VESR_EL2.AET, VESR_EL2.ExT);
    else
        fault = AsyncExternalAbort(parity, pe_error_state, extflag);

    ClearPendingVirtualSError();
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// =====

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);
    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH; // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;

    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```

Library pseudocode for aarch32/exceptions/debug/DebugException

```
constant bits(4) DebugException_Breakpoint = '0001';
constant bits(4) DebugException_BKPT      = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint = '1010';
```

Library pseudocode for aarch32/exceptions/exceptions/ AArch32.CheckAdvSIMDOrFPRegisterTraps

```
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =====
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

        if (tid0 == '1' && reg == '0000') // FPSID
            || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then // MVFRx
                if ELUsingAArch32(EL2) then
                    AArch32.SystemAccessTrap(M32_Hyp, 0x8); // Exception_AdvSIMDFPAccessTrap
                else
                    AArch64.AArch32SystemAccessTrap(EL2, 0x8); // Exception_AdvSIMDFPAccessTrap
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception exceptype)

    il_is_valid = TRUE;

    case exceptype of
        when Exception_Uncategorized      ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap            ec = 0x01;
        when Exception_CP15RITrap         ec = 0x03;
        when Exception_CP15RRTTrap        ec = 0x04;
        when Exception_CP14RITrap         ec = 0x05;
        when Exception_CP14DITrap         ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap ec = 0x07;
        when Exception_FPIDTrap           ec = 0x08;
        when Exception_PACTrap            ec = 0x09;
        when Exception_LDST64BTrap        ec = 0x0A;
        when Exception_CP14RRTTrap        ec = 0x0C;
        when Exception_BranchTarget       ec = 0x0D;
        when Exception_IllegalState       ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall     ec = 0x11;
        when Exception_HypervisorCall     ec = 0x12;
        when Exception_MonitorCall        ec = 0x13;
        when Exception_InstructionAbort   ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAlignment        ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort          ec = 0x24;
        when Exception_NV2DataAbort        ec = 0x25;
        when Exception_FPTrappedException ec = 0x28;
        otherwise                          Unreachable();

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;

    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';

    return (ec,il);
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

```
// AArch32.GeneralExceptionsToAArch64()
// =====
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\)) ||
            (EL2Enabled\(\) && !ELUsingAArch32\(EL2\) && HCR_EL2.TGE == '1'));
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// =====
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch32.ExceptionClass(exceptype);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if exceptype IN {Exception\_InstructionAbort, Exception\_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elseif exceptype == Exception\_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch32.ResetControlRegisters(boolean cold_reset);
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// =====
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
assert !assert HaveAArch64HighestELUsingAArch32();

// Enter the highest implemented Exception level in AArch32 state
if HaveEL(EL3) then
    AArch32.WriteMode(M32_Svc);
    SCR.NS = '0'; // Secure state
elseif HaveEL(EL2) then
    AArch32.WriteMode(M32_Hyp);
else
    AArch32.WriteMode(M32_Svc);

// Reset System registers in the coproc=0b111x encoding space and other system components // Reset
AArch32.ResetControlRegisters(cold_reset);
FPEXC.EN = '0';

// Reset all other PSTATE fields, including instruction set and endianness according to the
// SCTLR values produced by the above call to ResetControlRegisters()
PSTATE.<A,I,F> = '111'; // All asynchronous exceptions masked
PSTATE.IT = '00000000'; // IT block state reset
PSTATE.T = SCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
PSTATE.E = SCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
PSTATE.IL = '0'; // Clear Illegal Execution state bit

// All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
// below are UNKNOWN bitstrings after reset. In particular, the return information registers
// R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
// is impossible to return from a reset in an architecturally defined way.
AArch32.ResetGeneralRegisters();
AArch32.ResetSIMDFPRegisters();
AArch32.ResetSpecialRegisters();
ResetExternalDebugRegisters(cold_reset);

bits(32) rv; // IMPLEMENTATION DEFINED reset vector

if HaveEL(EL3) then
    if MVBAR<0> == '1' then // Reset vector in MVBAR
        rv = MVBAR<31:1>:'0';
    else
        rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
else
    rv = RVBAR<31:1>:'0';

// The reset vector must be correctly aligned
assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

boolean branch_conditional = FALSE;
BranchTo(rv, BranchType_RESET, branch_conditional);
```

Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

```
// ExcVectorBase()
// =====

bits(32) ExcVectorBase()
if SCTLR.V == '1' then // Hivecs selected, base = 0xFFFF0000
    return Ones(16):Zeros(16);
else
    return VBAR<31:5>:Zeros(5);
```

Library pseudocode for aarch32/exceptions/ieeefp/AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// =====

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64\(\) then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    FPEXC.DEX    = '1';
    FPEXC.TFV    = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>;           // IDF,IXF,UFF,OFF,DZF,I0F
    FPEXC<10:8>  = '111';                                   // VECITR is RES1

    AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

```
// AArch32.CallHypervisor()
// =====
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL\(EL2\);

    if !ELUsingAArch32\(EL2\) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCEException(immediate);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// =====
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate)

    if AArch32.CurrentCond\(\) != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64\(\) then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCEException(immediate);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCEException

```
// AArch32.TakeHVCEException()
// =====

AArch32.TakeHVCEException(bits(16) immediate)
    assert HaveEL\(EL2\) && ELUsingAArch32\(EL2\);

    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;

    exception = ExceptionSyndrome\(Exception\_HypervisorCall\);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCException

```
// AArch32.TakeSMCException()
// =====

AArch32.TakeSMCException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCException

```
// AArch32.TakeSVCException()
// =====

AArch32.TakeSVCException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_SupervisorCall);
        exception.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```


Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// =====
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return,
                    integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = HSCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// =====
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                 integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// =====
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                        integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTL.R.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTL.R.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEEnabled

```
// AArch32.CheckAdvSIMDOrFPEEnabled()
// =====
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEEnabled(boolean fpexc_check, boolean advsimd)
    if PSTATE.EL == EL0 && (!EL2EnabledHaveEL() || !(EL2) || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0'
        // The PE behaves as if FPEXC.EN is 1
        AArch64.CheckFPEEnabled();
        AArch64.CheckFPAdvSIMDEnabled();
    elseif PSTATE.EL == EL0 && EL2EnabledHaveEL() && !(EL2) && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'
        if fpexc_check && HCR_EL2.RW == '0' then
            fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
            if fpexc_en == '0' then UNDEFINED;
            AArch64.CheckFPEEnabledAArch64.CheckFPAdvSIMDEnabled();
        else
            cpacr_asedis = CPACR.ASEDIS;
            cpacr_cp10 = CPACR.cp10;

            if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
                if NSACR.cp10 == '0' then cpacr_cp10 = '00';

            if PSTATE.EL != EL2 then
                // Check if Advanced SIMD disabled in CPACR
                if advsimd && cpacr_asedis == '1' then UNDEFINED;

                // Check if access disabled in CPACR
                case cpacr_cp10 of
                    when '00' disabled = TRUE;
                    when '01' disabled = PSTATE.EL == EL0;
                    when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
                    when '11' disabled = FALSE;
                if disabled then UNDEFINED;

            // If required, check FPEXC enabled bit.
            if fpexc_check && FPEXC.EN == '0' then UNDEFINED;

            AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if HaveEL(EL2) && !IsSecure() then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;

            if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in HCPTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                exception.syndrome<24:20> = ConditionSyndrome();

            if advsimd then
                exception.syndrome<5> = '1';
            else
                exception.syndrome<5> = '0';
                exception.syndrome<3:0> = '1010';           // coproc field, always 0xA

            if PSTATE.EL == EL2 then
                AArch32.TakeUndefInstrException(exception);
            else
                AArch32.TakeHypTrapException(exception);

        if HaveEL(EL3) && !ELUsingAArch32(EL3) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

    return;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap

```
// AArch32.CheckForSMCUndefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;

    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCUndefOrTrap(Zeros(16));
    else
        route_to_hyp = (EL2) && !IsSecureEL2EnabledHaveEL() && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            exception = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(exception);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

```
// AArch32.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
  if HaveFGTExt() then
    route_to_el2 = FALSE;
    if PSTATE.EL == EL0 then
      route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
        (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

    if route_to_el2 then
      exception = ExceptionSyndrome(Exception_SupervisorCall);
      exception.syndrome<15:0> = immediate;
      bits(64) preferred_exception_return = ThisInstrAddr();
      vect_offset = 0x0;

      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWFXTrap

```
// AArch32.CheckForWFXTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFXTrap(bits(2) target_el, WFXType wfxtype)
  assert HaveEL(target_el);

  // Check for routing to AArch64
  if !ELUsingAArch32(target_el) then
    AArch64.CheckForWFXTrap(target_el, wfxtype);
    return;

  boolean is_wfe = wfxtype IN {WFXType_WFE, WFXType_WFET};
  case target_el of
    when EL1
      trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
    when EL2
      trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
    when EL3
      trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

  if trap then
    if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.WFXTrap(wfxtype, target_el);

    if target_el == EL3 then
      AArch32.TakeMonitorTrapException();
    elsif target_el == EL2 then
      exception = ExceptionSyndrome(Exception_WFXTrap);
      exception.syndrome<24:20> = ConditionSyndrome();

      case wfxtype of
        when WFXType_WFI
          exception.syndrome<0> = '0';
        when WFXType_WFE
          exception.syndrome<0> = '1';

      AArch32.TakeHypTrapException(exception);
    else
      AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// =====
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[.].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

    // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
    next_instr = AArch32.MemSingle[NextInstrAddr(), 2, AccType_IFETCH, TRUE];

    if next_instr IN {'11xxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxxx',
                     '01001xxxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
        // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
        // taken on the IT instruction or the next instruction. This is not reflected in
        // the pseudocode, which always takes the exception on the IT instruction. This
        // also does not take into account cases where the next instruction is UNPREDICTABLE.
        UNDEFINED;

    return;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

```
// AArch32.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// =====
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[.].SED);
    if setend_disabled == '1' then
        UNDEFINED;

    return;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrap

```
// AArch32.SystemAccessTrap()
// =====
// Trapped system register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
    (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrapSyndrome

```
// AArch32.SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;

    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTTrap);
        when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros();

    if exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTTrap, Exception_CP15RTTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss<13:10> = instr<19:16>;          // CRn, Reg in case of VMRS
        iss<8:5>   = instr<15:12>;          // Rt
        iss<9>     = '0';                   // RES0

        if exception.exceptype != Exception_FPIDTrap then // When trap is not for VMRS
            iss<19:17> = instr<7:5>;          // opc2
            iss<16:14> = instr<23:21>;        // opc1
            iss<4:1>   = instr<3:0>;          // CRm
        else //VMRS Access
            iss<19:17> = '000';              //opc2 - Hardcoded for VMRS
            iss<16:14> = '111';              //opc1 - Hardcoded for VMRS
            iss<4:1>   = '0000';             //CRm - Hardcoded for VMRS
        elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTrap} then
            // Trapped MRRC/MCRR, VMRS/VMSR
            iss<19:16> = instr<7:4>;          // opc1
            iss<13:10> = instr<19:16>;        // Rt2
            iss<8:5>   = instr<15:12>;        // Rt
            iss<4:1>   = instr<3:0>;          // CRm
        elsif exception.exceptype == Exception_CP14DTTTrap then
            // Trapped LDC/STC
            iss<19:12> = instr<7:0>;          // imm8
            iss<4>     = instr<23>;           // U
            iss<2:1>   = instr<24,21>;        // P,W
            if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
                iss<8:5> = bits(4) UNKNOWN;
                iss<3>   = '1';
            elsif exception.exceptype == Exception_Uncategorized then
                // Trapped for unknown reason
                iss<8:5> = instr<19:16>;      // Rn
                iss<3>   = '0';

            iss<0> = instr<20>;                // Direction

        exception.syndrome<24:20> = ConditionSyndrome();
        exception.syndrome<19:0>  = iss;

    return exception;
```


Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(exception);

// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x14;

    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// =====
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException()
    exception = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException(ExceptionRecord exception)

    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.UndefinedFault

```
// AArch32.UndefinedFault()
// =====

AArch32.UndefinedFault()

    if AArch32.GeneralExceptionsToAArch64\(\) then AArch64.UndefinedFault\(\);
    AArch32.TakeUndefInstrException\(\);
```

Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid

```
// AArch32.DomainValid()
// =====
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)
    assert statuscode != Fault\_None;

    case statuscode of
        when Fault\_Domain
            return TRUE;
        when Fault\_Translation, Fault\_AccessFlag, Fault\_SyncExternalOnWalk, Fault\_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusLD

```
// AArch32.FaultStatusLD()
// =====
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    bits(32) fsr = Zeros\(\);
    if HaveRASExt\(\) && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType\_DC, AccType\_IC,
                            AccType\_AT, AccType\_ATPAN} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '1';
    fsr<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return fsr;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusSD

```
// AArch32.FaultStatusSD()
// =====
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    bits(32) fsr = Zeros\(\);
    if HaveRASExt\(\) && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType\_DC, AccType\_IC,
            AccType\_AT, AccType\_ATPAN} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '0';
    fsr<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);
    if d_side then
        fsr<7:4> = fault.domain; // Domain field (data fault only)

    return fsr;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

```
// AArch32.FaultSyndrome()
// =====
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    bits(25) iss = Zeros\(\);

    if HaveRASExt\(\) && IsAsyncAbort(fault) then
        iss<11:10> = fault.errortype; // AET

    if d_side then
        if (IsSecondStage(fault) && !fault.s2fslwalk &&
            (!IsExternalSyncAbort(fault) ||
            (!HaveRASExt\(\) && fault.acctype == AccType\_TTW &&
            boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")))) then
            iss<24:14> = LSInstructionSyndrome();

        if fault.acctype IN {AccType\_DC, AccType\_IC, AccType\_AT, AccType\_ATPAN} then
            iss<8> = '1'; iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';

    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fslwalk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return iss;
```

Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =====
// Function that gives the Short-descriptor FSR code for different types of Fault
bits(5) EncodeSDFSC(Fault statuscode, integer level)

bits(5) result;
case statuscode of
  when Fault\_AccessFlag
    assert level IN {1,2};
    result = if level == 1 then '00011' else '00110';
  when Fault\_Alignment
    result = '00001';
  when Fault\_Permission
    assert level IN {1,2};
    result = if level == 1 then '01101' else '01111';
  when Fault\_Domain
    assert level IN {1,2};
    result = if level == 1 then '01001' else '01011';
  when Fault\_Translation
    assert level IN {1,2};
    result = if level == 1 then '00101' else '00111';
  when Fault\_SyncExternal
    result = '01000';
  when Fault\_SyncExternalOnWalk
    assert level IN {1,2};
    result = if level == 1 then '01100' else '01110';
  when Fault\_SyncParity
    result = '11001';
  when Fault\_SyncParityOnWalk
    assert level IN {1,2};
    result = if level == 1 then '11100' else '11110';
  when Fault\_AsyncParity
    result = '11000';
  when Fault\_AsyncExternal
    result = '10110';
  when Fault\_Debug
    result = '00010';
  when Fault\_TLBConflict
    result = '10000';
  when Fault\_Lockdown
    result = '10100'; // IMPLEMENTATION DEFINED
  when Fault\_Exclusive
    result = '10101'; // IMPLEMENTATION DEFINED
  when Fault\_ICacheMaint
    result = '00100';
  otherwise
    Unreachable\(\);

return result;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// =====

bits(32) A32ExpandImm(bits(12) imm12)

// PSTATE.C argument to following function call does not affect the imm32 result.
(imm32, -) = A32ExpandImm\_C(imm12, PSTATE.C);

return imm32;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm_C

```
// A32ExpandImm_C()
// =====

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRTYPE_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

Library pseudocode for aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// =====

(SRType, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

    case srtype of
        when '00'
            shift_t = SRTYPE_LSL; shift_n = UInt(imm5);
        when '01'
            shift_t = SRTYPE_LSR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRTYPE_ASR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRTYPE_RRX; shift_n = 1;
            else
                shift_t = SRTYPE_ROR; shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

Library pseudocode for aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// =====

SRType DecodeRegShift(bits(2) srtype)

    case srtype of
        when '00' shift_t = SRTYPE_LSL;
        when '01' shift_t = SRTYPE_LSR;
        when '10' shift_t = SRTYPE_ASR;
        when '11' shift_t = SRTYPE_ROR;

    return shift_t;
```

Library pseudocode for aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

Library pseudocode for aarch32/functions/common/RRX_C

```
// RRX_C()
// =====

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

Library pseudocode for aarch32/functions/common/SRType

```
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

Library pseudocode for aarch32/functions/common/Shift

```
// Shift()
// =====

bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
    (result, -) = Shift\_C(value, srtype, amount, carry_in);
    return result;
```

Library pseudocode for aarch32/functions/common/Shift_C

```
// Shift_C()
// =====

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
    assert !(srtype == SRType\_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType\_LSL
                (result, carry_out) = LSL\_C(value, amount);
            when SRType\_LSR
                (result, carry_out) = LSR\_C(value, amount);
            when SRType\_ASR
                (result, carry_out) = ASR\_C(value, amount);
            when SRType\_ROR
                (result, carry_out) = ROR\_C(value, amount);
            when SRType\_RRX
                (result, carry_out) = RRX\_C(value, carry_in);

    return (result, carry_out);
```

Library pseudocode for aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// =====

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm\_C(imm12, PSTATE.C);

    return imm32;
```

Library pseudocode for aarch32/functions/common/T32ExpandImm_C

```
// T32ExpandImm_C()
// =====

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

Library pseudocode for aarch32/functions/common/VCGEType

```
enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```

Library pseudocode for aarch32/functions/common/VFPNegMul

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
```

Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.
// Check for coarse-grained CP15 traps in HSTR and HCR.

boolean AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if PSTATE.EL == EL0 && !ELUsingAArch32(EL2) then
            return AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !(major IN {4,14}) && HSTR<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR.TIDCP
        if (HCR.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
             (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```

Library pseudocode for aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareability != Shareability_NSH then
            passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    return passed;
```

Library pseudocode for aarch32/functions/exclusive/AArch32.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

Library pseudocode for aarch32/functions/exclusive/AArch32.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```


Library pseudocode for aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)
    acctype = AccType\_ATOMIC;
    iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        MarkExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.address, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```

Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEEnabled

```
// CheckAdvSIMDOrVFPEEnabled()
// =====

CheckAdvSIMDOrVFPEEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// =====

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/CheckVFPEEnabled

```
// CheckVFPEEnabled()
// =====

CheckVFPEEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =====

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTYPE\_Infinity); inf2 = (type2 == FPTYPE\_Infinity);
        zero1 = (type1 == FPTYPE\_Zero); zero2 = (type2 == FPTYPE\_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr);
            FPProcessException(FPEXC\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

Library pseudocode for aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =====

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRTType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTYPE\_Infinity); inf2 = (type2 == FPTYPE\_Infinity);
        zero1 = (type1 == FPTYPE\_Zero); zero2 = (type2 == FPTYPE\_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0');
        result = FPHalvedSub(three, product, fpcr);
    return result;
```

Library pseudocode for aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =====

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRTType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPCRTType_Infinity); inf2 = (type2 == FPCRTType_Infinity);
        zero1 = (type1 == FPCRTType_Zero); zero2 = (type2 == FPCRTType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) two = FPTwo('0');
        result = FPSub(two, product, fpcr);
    return result;
```

Library pseudocode for aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// =====

FPCRTType StandardFPSCRValue()
    bits(32) upper = '00000000000000000000000000000000';
    bits(32) lower = '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '0000000000000000000000';
    return upper : lower;
```

Library pseudocode for aarch32/functions/memory/AArch32.CheckAlignment

```
// AArch32.CheckAlignment()
// =====

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype,
                                boolean iswrite)

    if PSTATE.EL == EL0 && !ELUsingAArch32(S1TranslationRegime()) then
        A = SCTLR[].A; //use AArch64 register, when higher Exception level is using AArch64
    elsif PSTATE.EL == EL2 then
        A = HSCTLR.A;
    else
        A = SCTLR.A;
    aligned = (address == Align(address, alignment));
    atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
                          AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC };
    vector = acctype == AccType_VEC;

    // AccType_VEC is used for SIMD element alignment checks only
    check = (atomic || ordered || vector || A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```



```

// AArch32.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned]
acctype, boolean wasaligned]
    boolean ispair = FALSE;
    return AArch32.MemSingle[address, size, acctype, aligned, ispair];
[address, size, acctype, wasaligned, ispair];

// AArch32.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean
acctype, boolean wasaligned, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc =

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    (memstatus, value) = value = _Mem[memaddrdesc, size, accdesc, FALSE];

    return value;

// AArch32.MemSingle[] - assignment (write) form
// =====
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =====

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value
acctype, boolean wasaligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
[address, size, acctype, wasaligned, ispair] = value;
    return;

// AArch32.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bi
acctype, boolean wasaligned, boolean ispair] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};

```

```

assert address == Align(address, size);(address, size);

AddressDescriptor memaddrdesc;
iswrite = TRUE;

memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch32.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability !=

AddressDescriptor memaddrdesc;
iswrite = TRUE;

memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch32.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH thenthen
    ClearExclusiveByAddress(memaddrdesc.paddress,
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);

memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
(acctype);

Mem[memaddrdesc, size, accdesc] = value;
return;

```

Library pseudocode for aarch32/functions/memory/Hint_PreloadData

```
Hint_PreloadData(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadDataForWrite

```
Hint_PreloadDataForWrite(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadInstr

```
Hint_PreloadInstr(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/MemA

```

// MemA[] - non-assignment form
// =====

bits(8*size) MemA(bits(32) address, integer size)
    acctype = AccType_ATOMIC;
    return Mem_with_type[address, size, acctype];

// MemA[] - assignment form
// =====

MemA(bits(32) address, integer size) = bits(8*size) value
    acctype = AccType_ATOMIC;
    Mem_with_type[address, size, acctype] = value;
    return;

```

Library pseudocode for aarch32/functions/memory/MemO

```
// MemO[] - non-assignment form
// =====

bits(8*size) MemO(bits(32) address, integer size]
    acctype = AccType\_ORDERED;
    return Mem\_with\_type[address, size, acctype];

// MemO[] - assignment form
// =====

MemO(bits(32) address, integer size] = bits(8*size) value
    acctype = AccType\_ORDERED;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemS

```
// MemS[] - non-assignment form
// =====
// Memory accessor for streaming load multiple instructions

bits(8*size) MemS(bits(32) address, integer size]
    acctype = AccType\_A32LSMD;
    return Mem\_with\_type[address, size, acctype];

// MemS[] - assignment form
// =====
// Memory accessor for streaming store multiple instructions

MemS(bits(32) address, integer size] = bits(8*size) value
    acctype = AccType\_A32LSMD;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemU

```
// MemU[] - non-assignment form
// =====

bits(8*size) MemU(bits(32) address, integer size]
    acctype = AccType\_NORMAL;
    return Mem\_with\_type[address, size, acctype];

// MemU[] - assignment form
// =====

MemU(bits(32) address, integer size] = bits(8*size) value
    acctype = AccType\_NORMAL;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemU_unpriv

```
// MemU_unpriv[] - non-assignment form
// =====

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType\_UNPRIV;
    return Mem\_with\_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// =====

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType\_UNPRIV;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```



```

// Mem_with_type[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    boolean ispair = FALSE;
    return Mem_with_type[address, size, acctype, ispair];

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    boolean iswrite = FALSE;
    integer halfsize = size DIV 2;

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then

        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned, ispair];

    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem_with_type[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
    boolean ispair = FALSE;
    Mem_with_type[address, size, acctype, ispair] = value;

Mem_with_type[bits(32) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value
    boolean iswrite = TRUE;
    integer halfsize = size DIV 2;

    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did

```

```

// not, so we must be changing to a new translation page.
c = ConstrainUnpredictable\(Unpredictable\_DEVPAGE2\);
assert c IN {Constraint\_FAULT, Constraint\_NONE};
if c == Constraint\_NONE then aligned = TRUE;

for i = 1 to size-1
    AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
else
    AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
return;

```

Library pseudocode for aarch32/functions/ras/AArch32.ESBOperation

```

// AArch32.ESBOperation()
// =====
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\);
if !route_to_aarch64 && EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
if !route_to_aarch64 && HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
    route_to_aarch64 = SCR_EL3.EA == '1';

if route_to_aarch64 then
    AArch64.ESBOperation\(\);
    return;

route_to_monitor = HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && SCR.EA == '1';
route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) && (HCR.TGE == '1' || HCR.AMO == '1');

if route_to_monitor then
    target = M32\_Monitor;
elseif route_to_hyp || PSTATE.M == M32\_Hyp then
    target = M32\_Hyp;
else
    target = M32\_Abort;

if IsSecure\(\) then
    mask_active = TRUE;
elseif target == M32\_Monitor then
    mask_active = SCR.AW == '1' && (!HaveEL\(EL2\) || (HCR.TGE == '0' && HCR.AMO == '0'));
else
    mask_active = target == M32\_Abort || PSTATE.M == M32\_Hyp;

mask_set = PSTATE.A == '1';
(-, el) = ELFromM32\(target\);
intdis = Halted\(\) || ExternalDebugInterruptsDisabled\(el\);
masked = intdis || (mask_active && mask_set);

// Check for a masked Physical SError pending that can be synchronized
// by an Error synchronization event.
if masked && IsSynchronizablePhysicalSErrorPending\(\) then
    syndrome32 = AArch32.PhysicalSErrorSyndrome\(\);
    DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
    ClearPendingPhysicalSError\(\);

return;

```

Library pseudocode for aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

```

// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();

```

Library pseudocode for aarch32/functions/ras/AArch32.ReportDeferredSError

```
// AArch32.ReportDeferredSError()
// =====
// Return deferred SError syndrome

bits(32) AArch32.ReportDeferredSError(bits(2) AET, bit ExT)
    bits(32) target;
    target<31> = '1'; // A
    syndrome = Zeros(16);
    if PSTATE.EL == EL2 then
        syndrome<11:10> = AET; // AET
        syndrome<9> = ExT; // EA
        syndrome<5:0> = '010001'; // DFSC
    else
        syndrome<15:14> = AET; // AET
        syndrome<12> = ExT; // ExT
        syndrome<9> = TTBCR.EAE; // LPAE
        if TTBCR.EAE == '1' then // Long-descriptor format
            syndrome<5:0> = '010001'; // STATUS
        else // Short-descriptor format
            syndrome<10,3:0> = '10110'; // FS
    if HaveAArch64HaveAnyAArch64() then
        target<24:0> = ZeroExtend(syndrome); // Any RES0 fields must be set to zero
    else
        target<15:0> = syndrome;
    return target;
```

Library pseudocode for aarch32/functions/ras/AArch32.SErrorSyndrome

```
type AArch32.SErrorSyndrome is (
    bits(2) AET,
    bit ExT
)
```

Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

```
// AArch32.vESBOperation()
// =====
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESBOperation();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a
    // virtual SError interrupt might be pending
    vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR.VA == '1';
    vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        HCR.VA = '0'; // Clear pending virtual SError

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// =====

AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN;    // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
        if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```
// AArch32.ResetSIMDFPRegisters()
// =====

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSpecialRegisters

```
// AArch32.ResetSpecialRegisters()
// =====

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_svc<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSR_und<31:0> = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

```
AArch32.ResetSystemRegisters(boolean cold_reset);
```

Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

```
// ALUExceptionReturn()
// =====

ALUExceptionReturn(bits(32) address)
  if PSTATE.EL == EL2 then
    UNDEFINED;
  elsif PSTATE.M IN {M32\_User,M32\_System} then
    Constraint c = ConstrainUnpredictable(Unpredictable\_ALUEXCEPTIONRETURN);
    assert c IN {Constraint\_UNDEF, Constraint\_NOP};
    case c of
      when Constraint\_UNDEF
        UNDEFINED;
      when Constraint\_NOPEndOfInstruction();
    else
      AArch32.ExceptionReturn(address, SPSR[]);
```

Library pseudocode for aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// =====

ALUWritePC(bits(32) address)
  if CurrentInstrSet() == InstrSet\_A32 then
    BXWritePC(address, BranchType\_INDIR);
  else
    BranchWritePC(address, BranchType\_INDIR);
```

Library pseudocode for aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// =====

BXWritePC(bits(32) address, BranchType branch_type)
  if address<0> == '1' then
    SelectInstrSet(InstrSet\_T32);
    address<0> = '0';
  else
    SelectInstrSet(InstrSet\_A32);
    // For branches to an unaligned PC counter in A32 state, the processor takes the branch
    // and does one of:
    // * Forces the address to be aligned
    // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
    if address<1> == '1' && ConstrainUnpredictableBool(Unpredictable\_A32FORCEALIGNPC) then
      address<1> = '0';
    boolean branch_conditional = AArch32.CurrentCond() != '111x';
    BranchTo(address, branch_type, branch_conditional);
```

Library pseudocode for aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// =====

BranchWritePC(bits(32) address, BranchType branch_type)
  if CurrentInstrSet() == InstrSet\_A32 then
    address<1:0> = '00';
  else
    address<0> = '0';
  boolean branch_conditional = AArch32.CurrentCond() != '111x';
  BranchTo(address, branch_type, branch_conditional);
```

Library pseudocode for aarch32/functions/registers/CBWritePC

```
// CBWritePC()
// =====
// Takes a branch from a CBNZ/CBZ instruction.

CBWritePC(bits(32) address)
    assert CurrentInstrSet\(\) == InstrSet\_T32;
    address<0> = '0';
    boolean branch_conditional = TRUE;
    BranchTo(address, BranchType\_DIR, branch_conditional);
```

Library pseudocode for aarch32/functions/registers/D

```
// D[] - non-assignment form
// =====

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2];
    return vreg<base+63:base>;

// D[] - assignment form
// =====

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2];
    vreg<base+63:base> = value;
    V[n DIV 2] = vreg;
    return;
```

Library pseudocode for aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// =====

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return \_Dclone[n];
```

Library pseudocode for aarch32/functions/registers/LR

```
// LR - assignment form
// =====

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// =====

bits(32) LR
    return R[14];
```

Library pseudocode for aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =====

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType\_INDIR);
```

Library pseudocode for aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// =====

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    case n of // Select index by mode:      usr fiq irq svc abt und hyp
        when 8      result = RBankSelect(mode, 8, 24, 8, 8, 8, 8, 8);
        when 9      result = RBankSelect(mode, 9, 25, 9, 9, 9, 9, 9);
        when 10     result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11     result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12     result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13     result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14     result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise   result = n;

    return result;
```

Library pseudocode for aarch32/functions/registers/Monitor_mode_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

Library pseudocode for aarch32/functions/registers/PC

```
// PC - non-assignment form
// =====

bits(32) PC
    return R[15]; // This includes the offset from AArch32 state
```

Library pseudocode for aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// =====

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

Library pseudocode for aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =====

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n];

// Q[] - assignment form
// =====

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n] = value;
    return;
```


Library pseudocode for aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// =====

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

Library pseudocode for aarch32/functions/registers/R

```
// R[] - assignment form
// =====

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =====

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet\_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];
```

Library pseudocode for aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =====

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
    integer svc, integer abt, integer und, integer hyp)

    case mode of
        when M32\_User      result = usr;    // User mode
        when M32\_FIQ       result = fiq;    // FIQ mode
        when M32\_IRQ       result = irq;    // IRQ mode
        when M32\_Svc       result = svc;    // Supervisor mode
        when M32\_Abort     result = abt;    // Abort mode
        when M32\_Hyp       result = hyp;    // Hyp mode
        when M32\_Undef     result = und;    // Undefined mode
        when M32\_System    result = usr;    // System mode uses User mode registers
        otherwise         Unreachable(); // Monitor mode

    return result;
```

Library pseudocode for aarch32/functions/registers/Rmode

```
// Rmode[] - non-assignment form
// =====

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookupRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =====

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if if ! HaveAArch64HighestELUsingAArch32() && ConstrainUnpredictableBool(Unpredictable_ZEROWRITE)
            _R[LookupRIndex(n, mode)] = ZeroExtend(value);
        else
            _R[LookupRIndex(n, mode)]<31:0> = value;

    return;
```

Library pseudocode for aarch32/functions/registers/S

```
// S[] - non-assignment form
// =====

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4];
    return vreg<base+31:base>;

// S[] - assignment form
// =====

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4];
    vreg<base+31:base> = value;
    V[n DIV 4] = vreg;
    return;
```

Library pseudocode for aarch32/functions/registers/SP

```
// SP - assignment form
// =====

SP = bits(32) value
  R[13] = value;
  return;

// SP - non-assignment form
// =====

bits(32) SP
  return R[13];
```

Library pseudocode for aarch32/functions/registers/_Dclone

```
array bits(64) _Dclone[0..31];
```

Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =====

AArch32.ExceptionReturn(bits(32) new_pc, bits(32) spsr)

  SynchronizeContext();
  // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
  // mechanism
  SetPSTATEFromPSR(spsr);
  ClearExclusiveLocal(ProcessorID());
  SendEventLocal();

  if PSTATE.IL == '1' then
    // If the exception return is illegal, PC[1:0] are UNKNOWN
    new_pc<1:0> = bits(2) UNKNOWN;
  else
    // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
    if PSTATE.T == '1' then
      new_pc<0> = '0'; // T32
    else
      new_pc<1:0> = '00'; // A32

  boolean branch_conditional = AArch32.CurrentCond() != '111x';
  BranchTo(new_pc, BranchType_ERET, branch_conditional);

  CheckExceptionCatch(FALSE); // Check for debug event on exception return
```

Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

```
// AArch32.ExecutingCP10or11Instr()
// =====

boolean AArch32.ExecutingCP10or11Instr()
  instr = ThisInstr();
  instr_set = CurrentInstrSet();
  assert instr_set IN {InstrSet_A32, InstrSet_T32};

  if instr_set == InstrSet_A32 then
    return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
  else // InstrSet_T32
    return (instr<31:28> == '111x' && (instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
```

Library pseudocode for aarch32/functions/system/AArch32.ITAdvance

```
// AArch32.ITAdvance()
// =====

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead

```
// Read from a 32-bit AArch32 System register and return the register's contents.
bits(32) AArch32.SysRegRead(integer cp_num, bits(32) instr);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead64

```
// Read from a 64-bit AArch32 System register and return the register's contents.
bits(64) AArch32.SysRegRead64(integer cp_num, bits(32) instr);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

```
// AArch32.SysRegReadCanWriteAPSR()
// =====
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn = UInt(instr<19:16>);
    CRm = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite

```
// Write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, bits(32) val);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite64

```
// Write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, bits(64) val);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWriteM

```
// Read a value from a virtual address and write it to an AArch32 System register.
AArch32.SysRegWriteM(integer cp_num, bits(32) instr, bits(32) address);
```

Library pseudocode for aarch32/functions/system/AArch32.WriteMode

```
// AArch32.WriteMode()
// =====
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M    = mode;
    PSTATE.EL   = el;
    PSTATE.nRW  = '1';
    PSTATE.SP   = (if mode IN {M32\_User,M32\_System} then '0' else '1');
    return;
```

Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

```
// AArch32.WriteModeByInstr()
// =====
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);

    // 'valid' is set to FALSE if 'mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32\_Hyp || mode == M32\_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32\_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```

Library pseudocode for aarch32/functions/system/BadMode

```
// BadMode()
// =====

boolean BadMode(bits(5) mode)
// Return TRUE if 'mode' encodes a mode that is not valid for this implementation
case mode of
  when M32\_Monitor
    valid = HaveAArch32EL\(EL3\);
  when M32\_Hyp
    valid = HaveAArch32EL\(EL2\);
  when M32\_FIQ, M32\_IRQ, M32\_Svc, M32\_Abort, M32\_Undef, M32\_System
    // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
    // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
    // AArch64, then these modes are EL1 modes.
    // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
    valid = HaveAArch32EL\(EL1\);
  when M32\_User
    valid = HaveAArch32EL\(EL0\);
  otherwise
    valid = FALSE;          // Passed an illegal mode value
return !valid;
```

Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

case SYSm of
  when '000xx', '00100' // R8_usr to R12_usr
    if mode != M32\_FIQ then UNPREDICTABLE;
  when '00101' // SP_usr
    if mode == M32\_System then UNPREDICTABLE;
  when '00110' // LR_usr
    if mode IN {M32\_Hyp,M32\_System} then UNPREDICTABLE;
  when '010xx', '0110x', '01110' // R8_fiq to R12_fiq, SP_fiq, LR_fiq
    if mode == M32\_FIQ then UNPREDICTABLE;
  when '1000x' // LR_irq, SP_irq
    if mode == M32\_IRQ then UNPREDICTABLE;
  when '1001x' // LR_svc, SP_svc
    if mode == M32\_Svc then UNPREDICTABLE;
  when '1010x' // LR_abt, SP_abt
    if mode == M32\_Abort then UNPREDICTABLE;
  when '1011x' // LR_und, SP_und
    if mode == M32\_Undef then UNPREDICTABLE;
  when '1110x' // LR_mon, SP_mon
    if !HaveEL\(EL3\) || !IsSecure\(\) || mode == M32\_Monitor then UNPREDICTABLE;
  when '11110' // ELR_hyp, only from Monitor or Hyp mode
    if !HaveEL\(EL2\) || !(mode IN {M32\_Monitor,M32\_Hyp}) then UNPREDICTABLE;
  when '11111' // SP_hyp, only from Monitor mode
    if !HaveEL\(EL2\) || mode != M32\_Monitor then UNPREDICTABLE;
  otherwise
    UNPREDICTABLE;

return;
```

Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// =====
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;          // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        if HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>;                // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);
    return;
```

Library pseudocode for aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =====

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

Library pseudocode for aarch32/functions/system/CurrentCond

```
bits(4) AArch32.CurrentCond();
```

Library pseudocode for aarch32/functions/system/InITBlock

```
// InITBlock()
// =====

boolean InITBlock()
    if CurrentInstrSet() == InstrSet\_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

Library pseudocode for aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// =====

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// =====

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    bits(32) new_spsr = SPSR[];

    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>; // N,Z,C,V,Q flags, IT[1:0],J bits

    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>; // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>; // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>; // I,F interrupt masks, T bit, Mode bits

    SPSR[] = new_spsr; // UNPREDICTABLE if User or System mode

    return;
```

Library pseudocode for aarch32/functions/system/SPSRAccessValid

```
// SPSRAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRAccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110' // SPSR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '10000' // SPSR_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '10010' // SPSR_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '10100' // SPSR_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110' // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100' // SPSR_mon
            if !HaveEL(EL3) || mode == M32_Monitor || !IsSecure() then UNPREDICTABLE;
        when '11110' // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```


Library pseudocode for aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// =====

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet\_A32, InstrSet\_T32};
    assert iset IN {InstrSet\_A32, InstrSet\_T32};

    PSTATE.T = if iset == InstrSet\_A32 then '0' else '1';

    return;
```

Library pseudocode for aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

Library pseudocode for aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// =====

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSat0(i, N);
    return result;
```

Library pseudocode for aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =====

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSat0(i, N);
    return result;
```

Library pseudocode for aarch32/ic/AArch32.IC

```
// AArch32.IC()
// =====
// Perform Instruction Cache Operation.

AArch32.IC(CacheOpScope opscope)
    regval = bits(32) UNKNOWN;
    AArch32.IC(regval, opscope);

// AArch32.IC()
// =====
// Perform Instruction Cache Operation.

AArch32.IC(bits(32) regval, CacheOpScope opscope)
    CacheRecord cache;
    AccType acctype = AccType_IC;

    cache.acctype = acctype;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        if opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR.FB == '1') then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = ZeroExtend(regval);
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;
        need_translate = ICIInstNeedsTranslation(opscope);

        cache.shareability = Shareability_NSH;
        cache.vaddress = ZeroExtend(regval);
        cache.translated = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        wasaligned = TRUE;
        iswrite = FALSE;
        size = 0;
        memaddrdesc = AArch32.TranslateAddress(regval, acctype, iswrite, wasaligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        cache.paddress = memaddrdesc.paddress;
        CACHE_OP(cache);
    return;
```



```

// AArch32.DefaultTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, without TEX remap

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX, bit C, bit B, bit S)
    MemoryAttributes memattrs;

// Reserved values map to allocated values
if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
    bits(5) texcb;
    (-, texcb) = ConstrainUnpredictableBits(Unpredictable\_RESTEXCB);
    TEX = texcb<4:2>; C = texcb<1>; B = texcb<0>;

// Distinction between Inner Shareable and Outer Shareable is not supported in this format
// A memory region is either Non-shareable or Outer Shareable
case TEX:C:B of
    when '00000'
        // Device-nGnRnE
        memattrs.memtype = MemType\_Device;
        memattrs.device = DeviceType\_nGnRnE;
        memattrs.shareability = Shareability\_OSH;
    when '00001', '01000'
        // Device-nGnRE
        memattrs.memtype = MemType\_Device;
        memattrs.device = DeviceType\_nGnRE;
        memattrs.shareability = Shareability\_OSH;
    when '00010'
        // Write-through Read allocate
        memattrs.memtype = MemType\_Normal;
        memattrs.inner.attrs = MemAttr\_WT;
        memattrs.inner.hints = MemHint\_RA;
        memattrs.outer.attrs = MemAttr\_WT;
        memattrs.outer.hints = MemHint\_RA;
        memattrs.shareability = if S == '1' then Shareability\_OSH else Shareability\_NSH;
    when '00011'
        // Write-back Read allocate
        memattrs.memtype = MemType\_Normal;
        memattrs.inner.attrs = MemAttr\_WB;
        memattrs.inner.hints = MemHint\_RA;
        memattrs.outer.attrs = MemAttr\_WB;
        memattrs.outer.hints = MemHint\_RA;
        memattrs.shareability = if S == '1' then Shareability\_OSH else Shareability\_NSH;
    when '00100'
        // Non-cacheable
        memattrs.memtype = MemType\_Normal;
        memattrs.inner.attrs = MemAttr\_NC;
        memattrs.outer.attrs = MemAttr\_NC;
        memattrs.shareability = Shareability\_OSH;
    when '00110'
        memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    when '00111'
        // Write-back Read and Write allocate
        memattrs.memtype = MemType\_Normal;
        memattrs.inner.attrs = MemAttr\_WB;
        memattrs.inner.hints = MemHint\_RWA;
        memattrs.outer.attrs = MemAttr\_WB;
        memattrs.outer.hints = MemHint\_RWA;
        memattrs.shareability = if S == '1' then Shareability\_OSH else Shareability\_NSH;
    when '1xxxx'
        // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
        memattrs.memtype = MemType\_Normal;
        memattrs.inner = DecodeSDFAttr(C:B);
        memattrs.outer = DecodeSDFAttr(TEX<1:0>);

        if memattrs.inner.attrs == MemAttr\_NC && memattrs.outer.attrs == MemAttr\_NC then
            memattrs.shareability = Shareability\_OSH;
        else
            memattrs.shareability = if S == '1' then Shareability\_OSH else Shareability\_NSH;
    otherwise
        // Reserved, handled above

```

```
    Unreachable\(\);  
  
    // The Transient hint is not supported in this format  
    memattrs.inner.transient = FALSE;  
    memattrs.outer.transient = FALSE;  
    memattrs.tagged          = FALSE;  
  
    if memattrs.inner.attrs == MemAttr\_WB && memattrs.outer.attrs == MemAttr\_WB then  
        memattrs.xs = '0';  
    else  
        memattrs.xs = '1';  
  
    return memattrs;
```

Library pseudocode for aarch32/translation/attrs/AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, with TEX remap

MemoryAttributes AArch32.RemappedTEXDecode(bits(3) TEX, bit C, bit B, bit S)

    MemoryAttributes memattrs;

    region = UInt(TEX<0>:C:B);          // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        return MemoryAttributes IMPLEMENTATION_DEFINED;

    base = 2 * region;
    attrfield = PRRR<base+1:base>;

    if attrfield == '11' then          // Reserved, maps to allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESRRR);

    case attrfield of
        when '00'                      // Device-nGnRnE
            memattrs.memtype            = MemType_Device;
            memattrs.device             = DeviceType_nGnRnE;
            memattrs.shareability      = Shareability_OSH;
        when '01'                      // Device-nGnRE
            memattrs.memtype            = MemType_Device;
            memattrs.device             = DeviceType_nGnRE;
            memattrs.shareability      = Shareability_OSH;
        when '10'
            NSn = if S == '0' then PRRR.NS0 else PRRR.NS1;
            NOSm = PRRR<region+24> AND NSn;
            IRn = NMRR<base+1:base>;
            ORn = NMRR<base+17:base+16>;

            memattrs.memtype = MemType_Normal;
            memattrs.inner   = DecodeSDFAttr(IRn);
            memattrs.outer   = DecodeSDFAttr(ORn);
            if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
                memattrs.shareability = Shareability_OSH;
            else
                bits(2) sh = NSn:NOSm;
                memattrs.shareability = DecodeShareability(sh);
    (<NSn:NOSm>);
        when '11'
            Unreachable();

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tagged          = FALSE;

    if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_WB then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';

    return memattrs;
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

```
// AArch32.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert size IN {2,4};

    match = FALSE;
    mismatch = FALSE;

    for i = 0 to UInt\(DBGDIDR.BRPs\)
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint\(\) then
        reason = DebugHalt\_Breakpoint;
        Halt(reason);
    elseif (match || mismatch) then
        acctype = AccType\_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException\_Breakpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

```
// AArch32.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = NoFault();

    d_side = (acctype != AccType\_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool\(Unpredictable\_BPVECTORCATCHPRI\);

    if !d_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(vaddress, size);

    if fault.statuscode == Fault\_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch32.CheckBreakpoint(vaddress, size);

    if fault.statuscode == Fault\_None && !d_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(vaddress, size);

    return fault;
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = AArch32.VCRMATCH(vaddress);
    if size == 4 && !match && AArch32.VCRMATCH(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);

    if match then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_VectorCatch;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
                                     boolean iswrite, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    if acctype IN {AccType_TTW, AccType_IC, AccType_AT, AccType_ATPAN} then
        return NoFault();
    if acctype == AccType_DC then
        if !iswrite then
            return NoFault();
        elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
            return NoFault();

    match = FALSE;
    ispriv = AArch32.AccessUsesEL(acctype) != EL0;

    for i = 0 to UInt(DBGDIDR.WRPs)
        if AArch32.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        EDWAR = ZeroExtend(vaddress);
        Halt(reason);
    elsif match then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```


Library pseudocode for aarch32/translation/faults/AArch32.DebugFault

```
// AArch32.DebugFault()
// =====
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)
    FaultRecord fault;

    fault.statuscode = Fault_Debug;
    fault.acctype     = acctype;
    fault.write       = iswrite;
    fault.debugmoe    = debugmoe;
    fault.secondstage = FALSE;
    fault.s2fslwalk   = FALSE;

    return fault;
```

Library pseudocode for aarch32/translation/faults/AArch32.IPAIsOutOfRange

```
// AArch32.IPAIsOutOfRange()
// =====
// Check intermediate physical address bits not resolved by translation are ZERO

boolean AArch32.IPAIsOutOfRange(S2TTWParams walkparams, bits(40) ipa)
    // Input Address size
    iasize = AArch32.S2IASize(walkparams.t0sz);

    return iasize < 40 && !IsZero(ipa<39:iasize>);
```

Library pseudocode for aarch32/translation/faults/AArch32.S1HasAlignmentFault

```
// AArch32.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S1HasAlignmentFault(AccType acctype, boolean aligned,
                                     bit ntlsmid, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;

    if acctype == AccType_A32LSMD && ntlsmid == '0' && memattrs.device != DeviceType_GRE then
        return TRUE;

    return !aligned || acctype == AccType_DCZVA;
```



```

// AArch32.S1LDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

boolean AArch32.S1LDHasPermissionsFault(Regime regime, S1TTWParams walkparams,
                                         Permissions perms, MemType memtype,
                                         PASpace paspace, boolean ispriv,
                                         AccType acctype, boolean iswrite)

if HasUnprivileged(regime) then
    // Apply leaf permissions
    case perms.ap<2:1> of
        when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
        when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
        when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // R0 at PL1 only
        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 at any PL

    // Apply hierarchical permissions
    case perms.ap_table of
        when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
        when '01' (pr,pw,ur,uw) = ( pr, pw, '0','0'); // Privileged access
        when '10' (pr,pw,ur,uw) = ( pr, '0', ur, '0'); // Read-only
        when '11' (pr,pw,ur,uw) = ( pr, '0', '0','0'); // Read-only, privileged access

    wxn = walkparams.wxn;
    uwxn = walkparams.uwxn;
    xn = perms.xn OR perms.xn_table;
    pxn = perms.pxn OR perms.pxn_table;

    ux = ur AND NOT(xn OR (uw AND wxn));
    px = pr AND NOT(xn OR pxn OR (pw AND wxn) OR (uw AND uwxn));

    pan_access = !(acctype IN {AccType\_DC, AccType\_IFETCH, AccType\_AT});
    if HavePANExt() && pan_access then
        pan = PSTATE.PAN AND (ur OR uw);
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if IsSecure() && paspace == PAS\_NonSecure then
        x = x AND NOT(walkparams.sif);
else
    // Apply leaf permissions
    case perms.ap<2> of
        when '0' (r,w) = ('1','1'); // No effect
        when '1' (r,w) = ('1','0'); // Read-only

    // Apply hierarchical permissions
    case perms.ap_table<1> of
        when '0' (r,w) = ( r , w ); // No effect
        when '1' (r,w) = ( r , '0'); // Read-only

    xn = perms.xn OR perms.xn_table;
    x = NOT(xn OR (w AND walkparams.wxn));

if acctype == AccType\_IFETCH then
    constraint = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
    if constraint == Constraint\_FAULT && memtype == MemType\_Device then
        return TRUE;
    else
        return x == '0';
elseif acctype IN {AccType\_IC, AccType\_DC} then
    return FALSE;
elseif iswrite then
    return w == '0';
else
    return r == '0';

```

Library pseudocode for aarch32/translation/faults/AArch32.S1SDHasPermissionsFault

```
// AArch32.S1SDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

boolean AArch32.S1SDHasPermissionsFault(Permissions perms, MemType memtype,
                                         PASpace paspace, boolean ispriv,
                                         AccType acctype, boolean iswrite)

wxn = SCTL.R.WXN;
uwxn = SCTL.R.UWXN;
if SCTL.R.AFE == '0' then
    // Map Reserved encoding '100'
    if perms.ap == '100' then
        perms.ap = bits(3) IMPLEMENTATION_DEFINED "Reserved short descriptor AP encoding";

    case perms.ap of
        when '000' (pr,pw,ur,uw) = ('0','0','0','0'); // No access
        when '001' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
        when '010' (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1, R0 at PL0
        when '011' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
        // '100' is reserved
        when '101' (pr,pw,ur,uw) = ('1','0','0','0'); // R0 at PL1 only
        when '110' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 at any PL (deprecated)
        when '111' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 at any PL
    else // Simplified access permissions model
        case perms.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // R0 at PL1 only
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 at any PL

    ux = ur AND NOT(perms.xn OR (uw AND wxn));
    px = pr AND NOT(perms.xn OR perms.pxn OR (pw AND wxn) OR (uw AND uwxn));

    pan_access = !(acctype IN {AccType\_DC, AccType\_IFETCH, AccType\_AT});
    if HavePANExt() && pan_access then
        pan = PSTATE.PAN AND (ur OR uw);
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);

// Prevent execution from Non-secure space by PE in Secure state if SIF is set
if IsSecure() && paspace == PAS\_NonSecure then
    x = x AND NOT(if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF);

if acctype == AccType\_IFETCH then
    constraint = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
    if constraint == Constraint\_FAULT && memtype == MemType\_Device then
        return TRUE;
    else
        return x == '0';
elsif acctype IN {AccType\_IC, AccType\_DC} then
    return FALSE;
elsif iswrite then
    return w == '0';
else
    return r == '0';
```

Library pseudocode for aarch32/translation/faults/AArch32.S2HasAlignmentFault

```
// AArch32.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S2HasAlignmentFault(AccType acctype, boolean aligned,
                                   MemoryAttributes memattrs)
    if acctype == AccType\_IFETCH || memattrs.memtype != MemType\_Device then
        return FALSE;

    return !aligned || acctype == AccType\_DCZVA;
```

Library pseudocode for aarch32/translation/faults/AArch32.S2HasPermissionsFault

```
// AArch32.S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(boolean s2fslwalk, S2TTWParams walkparams,
                                       Permissions perms, MemType memtype,
                                       boolean ispriv, AccType acctype,
                                       boolean iswrite)

    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    if HaveExtendedExecuteNeverExt() then
        case perms.s2xn:perms.s2xn of
            when '00' (px, ux) = ( r , r );
            when '01' (px, ux) = ( '0', r );
            when '10' (px, ux) = ( '0', '0' );
            when '11' (px, ux) = ( r , '0' );

        x = if ispriv then px else ux;
    else
        x = r AND NOT(perms.s2xn);

    if s2fslwalk && walkparams.ptw == '1' && memtype == MemType\_Device then
        return TRUE;
    elsif acctype == AccType\_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
        if constraint == Constraint\_FAULT && memtype == MemType\_Device then
            return TRUE;
        else
            return x == '0';
    elsif acctype IN {AccType\_IC, AccType\_DC} then
        return FALSE;
    elsif iswrite then
        return w == '0';
    else
        return r == '0';
```

Library pseudocode for aarch32/translation/faults/AArch32.S2InconsistentSL

```
// AArch32.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

boolean AArch32.S2InconsistentSL(S2TTWParams walkparams)
    startlevel = AArch32.S2StartLevel(walkparams.sl0);
    levels     = FINAL\_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits    // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch32.S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

Library pseudocode for aarch32/translation/faults/AArch32.VAIsOutOfRange

```
// AArch32.VAIsOutOfRange()
// =====
// Check virtual address bits not resolved by translation are identical
// and of accepted value

boolean AArch32.VAIsOutOfRange(Regime regime, S1TTWParams walkparams, bits(32) va)
    if regime == Regime\_EL2 then
        // Input Address size
        iasize = AArch32.S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va<31:iasize>);
    elseif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        lo_iasize = AArch32.S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        up_iasize = AArch32.S1IASize(walkparams.t1sz);
        return !IsZero(va<31:lo_iasize>) && !IsOnes(va<31:up_iasize>);
    else
        return FALSE;
```

Library pseudocode for aarch32/translation/translation/AArch32.AccessUsesEL

```
// AArch32.AccessUsesEL()
// =====
// Determine the privilege associated with the access

bits(2) AArch32.AccessUsesEL(AccType acctype)
    if acctype == AccType\_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;
```

Library pseudocode for aarch32/translation/translation/AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =====
// Perform address translation as specified by VMSA-A32

AddressDescriptor AArch32.FullTranslate(bits(32) va, AccType acctype,
                                       boolean iswrite, boolean aligned)

    // Prepare fault fields in case a fault is detected
    fault = NoFault();
    fault.acctype = acctype;
    fault.write = iswrite;

    regime = TranslationRegime(PSTATE.EL, acctype);

    // First Stage Translation

    if regime == Regime_EL2 || TTBCR.EAE == '1' then
        (fault, ipa) = AArch32.S1TranslateLD(fault, regime, va, acctype,
                                             aligned, iswrite);
    else
        (fault, ipa, -) = AArch32.S1TranslateSD(fault, regime, va, acctype,
                                             aligned, iswrite);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(ZeroExtend(va), fault);

    if regime == Regime_EL10 && EL2Enabled() then
        ipa.vaddress = ZeroExtend(va);
        s2fslwalk = FALSE;
        (fault, pa) = AArch32.S2Translate(fault, ipa, s2fslwalk, acctype,
                                         aligned, iswrite);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(ZeroExtend(va), fault);
        else
            return pa;
    else
        return ipa;
```

Library pseudocode for aarch32/translation/translation/AArch32.OutputDomain

```
// AArch32.OutputDomain()
// =====
// Determine the domain the translated output address

bits(2) AArch32.OutputDomain(bits(4) domain)
    index = 2 * UInt(domain);
    Dn = DACR<index+1:index>;

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits(Unpredictable_RESDACR);

    return Dn;
```

Library pseudocode for aarch32/translation/translation/AArch32.S1DisabledOutput

```
// AArch32.S1DisabledOutput()
// =====
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes
(FaultRecord, AddressDescriptor) AArch32.S1DisabledOutput(FaultRecord fault,
    Regime regime, bits(32) va, AccType acctype, boolean aligned)

// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

MemoryAttributes memattrs;
if regime == Regime_EL10 && EL2Enabled() then
    default_cacheable = if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC;
else
    default_cacheable = '0';

if default_cacheable == '1' then
    // Use default cacheable settings
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WB;
    memattrs.inner.hints = MemHint_RWA;
    memattrs.outer.attrs = MemAttr_WB;
    memattrs.outer.hints = MemHint_RWA;
    memattrs.shareability = Shareability_NSH;
    if !ELUsingAArch32(EL2) && HaveMTE2Ext() then
        memattrs.tagged = HCR_EL2.DCT == '1';
    else
        memattrs.tagged = FALSE;
elseif acctype == AccType_IFETCH then
    // Instruction cacheability controlled by SCTLR/HSCTLR.I
    icache_en = if regime == Regime_EL2 then HSCTLR.I else SCTLR.I;

    memattrs.memtype = MemType_Normal;
    memattrs.shareability = Shareability_OSH;
    memattrs.tagged = FALSE;
    if icache_en == '1' then
        memattrs.inner.attrs = MemAttr_WT;
        memattrs.inner.hints = MemHint_RA;
        memattrs.outer.attrs = MemAttr_WT;
        memattrs.outer.hints = MemHint_RA;
    else
        memattrs.inner.attrs = MemAttr_NC;
        memattrs.outer.attrs = MemAttr_NC;
else
    // Treat memory region as Device
    memattrs.memtype = MemType_Device;
    memattrs.device = DeviceType_nGnRnE;
    memattrs.shareability = Shareability_OSH;
    memattrs.tagged = FALSE;

if HaveTrapLoadStoreMultipleDeviceExt() then
    ntlsmd = if regime == Regime_EL2 then HSCTLR.nTLSMD else SCTLR.nTLSMD;
else
    ntlsmd = '1';

if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmd, memattrs) then
    fault.statuscode = Fault_Alignment;
    return (fault, AddressDescriptor UNKNOWN);

FullAddress oa;
oa.address = ZeroExtend(va);
oa.paspace = if IsSecure() then PAS_Secure else PAS_NonSecure;
ipa = CreateAddressDescriptor(ZeroExtend(va), oa, memattrs);

return (fault, ipa);
```


Library pseudocode for aarch32/translation/translation/AArch32.S1Enabled

```
// AArch32.S1Enabled()
// =====
// Returns whether stage 1 translation is enabled for the active translation regime

boolean AArch32.S1Enabled(Regime regime)
    if regime == Regime\_EL2 then
        return HSCTLR.M == '1';
    elsif regime == Regime\_EL30 || !EL2Enabled\(\) then
        return SCTLR.M == '1';
    elsif ELUsingAArch32\(EL2\) then
        return HCR.<TGE,DC> == '00' && SCTLR.M == '1';
    else
        return HCR_EL2.<TGE,DC> == '00' && SCTLR.M == '1';
```



```

// AArch32.S1TranslateLD()
// =====
// Perform a stage 1 translation using long-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor)(FaultRecord, AddressDescriptor) AArch32.S1TranslateLD(FaultRecord fault,
    Regime regime, bits(32) va, AccType acctype, boolean aligned, boolean iswrite)

    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    if !AArch32.S1Enabled(regime) then
        return AArch32.S1DisabledOutput(fault, regime, va, acctype, aligned);

    walkparams = AArch32.S1DisabledOutput(fault, regime, va, acctype, aligned);

    walkparams = AArch32.GetS1TTWParams(regime, va);

    if AArch32.VAIsOutOfRange(regime, walkparams, va) then
        fault.level = 1;
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN);

    (fault, walkstate) = AArch32.S1WalkLD(fault, regime, walkparams, va);

    if fault.statuscode != AddressDescriptor UNKNOWN;

    (fault, walkstate) = AArch32.S1WalkLD(fault, regime, walkparams, va);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    ispriv = AddressDescriptor UNKNOWN;

    ispriv = AArch32.AccessUsesEL(acctype) != EL0;
    SetInGuardedPage(FALSE); // AArch32-VMVA does not guard any pages

    if AArch32.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmid,
        walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elseif IsAtomicRW(acctype) then
        if AArch32.S1LDHasPermissionsFault(regime, walkparams,
            walkstate.permissions,
            walkstate.memattrs.memtype,
            walkstate.baseaddress.paspace,
            ispriv, acctype, FALSE) then
            // The permission fault was not caused by lack of write permissions
            fault.statuscode = AArch32.S1LDHasPermissionsFault(regime, walkparams,
                walkstate.permissions,
                walkstate.memattrs.memtype,
                walkstate.baseaddress.paspace,
                ispriv, acctype, FALSE) then
            // The permission fault was not caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write = FALSE;
        elseif AArch32.S1LDHasPermissionsFault(regime, walkparams,
            walkstate.permissions,
            walkstate.memattrs.memtype,
            walkstate.baseaddress.paspace,
            ispriv, acctype, TRUE) then
            // The permission fault was caused by lack of write permissions
            fault.statuscode = AArch32.S1LDHasPermissionsFault(regime, walkparams,
                walkstate.permissions,
                walkstate.memattrs.memtype,
                walkstate.baseaddress.paspace,
                ispriv, acctype, TRUE) then
            // The permission fault was caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write = TRUE;
        elseif AArch32.S1LDHasPermissionsFault(regime, walkparams,

```

```

        walkstate.permissions,
        walkstate.memattrs.memtype,
        walkstate.baseaddress.paspace,
        ispriv, acctype, iswrite) then
    fault.statuscode = AArch32.S1LDHasPermissionsFault(regime, walkparams,
        walkstate.permissions,
        walkstate.memattrs.memtype,
        walkstate.baseaddress.paspace,
        ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;

    if fault.statuscode != Fault_None then
        return (fault, return (fault, AddressDescriptor UNKNOWN));

    icache_en = if regime == AddressDescriptor UNKNOWN);

    icache_en = if regime == Regime_EL2 then HSCTLR.I else SCTLR.I;
    dcache_en = if regime == Regime_EL2 then HSCTLR.C else SCTLR.C;

    if ((acctype == AccType_IFETCH &&
        (walkstate.memattrs.memtype == MemType_Device || icache_en == '0')) ||
        (acctype != AccType_IFETCH &&
        walkstate.memattrs.memtype == MemType_Normal && dcache_en == '0')) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    else
        memattrs = walkstate.memattrs;

    if (regime == Regime_EL10 && EL2Enabled() &&
        (if ELUsingAArch32(EL2) then HCR.VM else HCR_EL2.VM) == '1') then
        // Shareability of target memory subject to stage 2 translation
        // is maintained as input to stage 2
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = elsif (memattrs.memtype == NormaliseShareabilityMemType_Device(memattr

// Output Address
oa = ||
    (memattrs.inner.attrs == Stage0AMemAttr_NC(walkstate.baseaddress, && memattrs.outer.attrs == M
    // If the target memory is Device memory or is Normal memory
    // inner Non-Cacheable outer Non-Cacheable then it is forced to Outer Shareable
    memattrs.shareability = Shareability_OSH;

// Output Address
oa = Stage0A(walkstate.baseaddress, ZeroExtend(va), walkparams.tgx, walkstate.level);
ipa = CreateAddressDescriptor(va, walkparams.tgx, walkstate.level);
ipa = CreateAddressDescriptor(ZeroExtend(va), oa, memattrs);

return (fault, ipa);

```



```

// AArch32.S1TranslateSD()
// =====
// Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor, SDType)(FaultRecord, AddressDescriptor, AArch32.S1TranslateSD() AArch32
    Regime regime, bits(32) va, AccType acctype, boolean aligned, boolean iswrite)

    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    if !AArch32.S1Enabled(regime) then
        (fault, ipa) = (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, va, acctype, aligned)
        return (fault, ipa, AArch32.S1DisabledOutput(fault, regime, va, acctype, aligned));
        return (fault, ipa, SDType UNKNOWN);

    (fault, walkstate) = (fault, walkstate) = AArch32.S1WalkSD(fault, regime, va);

    if fault.statuscode != AArch32.S1WalkSD(fault, regime, va);

    if fault.statuscode != Fault_None then
        return (fault, return (fault, AddressDescriptor UNKNOWN, AddressDescriptor UNKNOWN, SDType

    ispriv = AArch32.AccessUsesEL(acctype) != EL0;
    domain = AArch32.OutputDomain(walkstate.domain);
    SetInGuardedPage(FALSE); // AArch32-VMVA does not guard any pages

    ntlsmd = if HaveTrapLoadStoreMultipleDeviceExt() then SCTLR.nTSLMD else '1';

    if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmd, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elseif !(acctype IN {AccType_IC, AccType_DC}) && domain == Domain_NoAccess then
        fault.statuscode = Fault_Domain;
    elseif domain == Domain_Client then
        if IsAtomicRW(acctype) then
            if AArch32.S1SDHasPermissionsFault(walkstate.permissions,
                walkstate.memattrs.memtype,
                walkstate.baseaddress.paspace,
                ispriv, acctype, FALSE) then
                // The permission fault was not caused by lack of write permissions
                fault.statuscode = AArch32.S1SDHasPermissionsFault(walkstate.permissions,
                    walkstate.memattrs.memtype,
                    walkstate.baseaddress.paspace,
                    ispriv, acctype, FALSE) then
                // The permission fault was not caused by lack of write permissions
                fault.statuscode = Fault_Permission;
                fault.write = FALSE;
            elseif AArch32.S1SDHasPermissionsFault(walkstate.permissions,
                walkstate.memattrs.memtype,
                walkstate.baseaddress.paspace,
                ispriv, acctype, TRUE) then
                // The permission fault was caused by lack of write permissions
                fault.statuscode = AArch32.S1SDHasPermissionsFault(walkstate.permissions,
                    walkstate.memattrs.memtype,
                    walkstate.baseaddress.paspace,
                    ispriv, acctype, TRUE) then
                // The permission fault was caused by lack of write permissions
                fault.statuscode = Fault_Permission;
                fault.write = TRUE;
            elseif AArch32.S1SDHasPermissionsFault(walkstate.permissions,
                walkstate.memattrs.memtype,
                walkstate.baseaddress.paspace,
                ispriv, acctype, iswrite) then
                fault.statuscode = AArch32.S1SDHasPermissionsFault(walkstate.permissions,
                    walkstate.memattrs.memtype,
                    walkstate.baseaddress.paspace,
                    ispriv, acctype, iswrite) then
                fault.statuscode = Fault_Permission;

    if fault.statuscode != Fault_None then

```

```

    fault.domain = walkstate.domain;
    return (fault, return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype););
if ((acctype == AddressDescriptor UNKNOWN, walkstate.sdftype);
if ((acctype == AccType_IFETCH &&
    (walkstate.memattrs.memtype == MemType_Device || SCTL.I == '0')) ||
    (acctype != AccType_IFETCH &&
    walkstate.memattrs.memtype == MemType_Normal && SCTL.C == '0')) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
else
    memattrs = walkstate.memattrs;

if (regime == Regime_EL10 && EL2Enabled() &&
    (if ELUsingAArch32(EL2) then HCR.VM else HCR_EL2.VM) == '1') then
    // Shareability of target memory subject to stage 2 translation
    // is maintained as input to stage 2
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = elseif (memattrs.memtype == NormaliseShareabilityMemType_Device(memattr

// Output Address
oa = ++
    (memattrs.inner.attrs == AArch32.SDStage0AMemAttr_NC(walkstate.baseaddress, va, walkstate.sd
ipa = && memattrs.outer.attrs == )) then
    // If the target memory is Device memory or is Normal memory
    // inner Non-Cacheable outer Non-Cacheable then it is forced to Outer Shareable
    memattrs.shareability = Shareability_OSHCreateAddressDescriptorMemAttr_NC(
// Output Address
oa = AArch32.SDStage0A(walkstate.baseaddress, va, walkstate.sdftype);
ipa = CreateAddressDescriptor(ZeroExtend(va), oa, memattrs);

return (fault, ipa, walkstate.sdftype);

```



```

// AArch32.S2Translate()
// =====
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor)(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault,
AddressDescriptor ipa, boolean s2fslwalk, AArch32.S2Translate(FaultRecord fault,
AddressDescriptor ipa, boolean s2fslwalk, AccType acctype,
boolean aligned, boolean iswrite)

assert IsZero(ipa.paddress.address<51:40>);

if !ELUsingAArch32(EL2) then
    slaarch64 = FALSE;
    return return AArch64.S2Translate(fault, ipa, slaarch64, s2fslwalk, acctype,
aligned, iswrite);

// Prepare fault fields in case a fault is detected
fault.statuscode = AArch64.S2Translate(fault, ipa, slaarch64, s2fslwalk, acctype,
aligned, iswrite);

// Prepare fault fields in case a fault is detected
fault.statuscode = Fault_None;
fault.secondstage = TRUE;
fault.s2fslwalk = s2fslwalk;
fault.ipaddress = ipa.paddress;

walkparams = AArch32.GetS2TTWParams();

if walkparams.vm == '0' then
    // Stage 2 is disabled
    return (fault, ipa);

if AArch32.IPAIsOutOfRange(walkparams, ipa.paddress.address<39:0>) then
    fault.statuscode = Fault_Translation;
    fault.level = 1;
    return (fault, return (fault, AddressDescriptor UNKNOWN);

(fault, walkstate) = AArch32.S2Walk(fault, walkparams, ipa);

if fault.statuscode != AddressDescriptor UNKNOWN);

(fault, walkstate) = AArch32.S2Walk(fault, walkparams, ipa);

if fault.statuscode != Fault_None then
    return (fault, return (fault, AddressDescriptor UNKNOWN);

ispriv = AddressDescriptor UNKNOWN);

ispriv = AArch32.AccessUsesEL(acctype) != EL0;

if AArch32.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elseif IsAtomicRW(acctype) then
    assert !s2fslwalk; // AArch32 does not support HW update of TT
    if AArch32.S2HasPermissionsFault(s2fslwalk, walkparams,
walkstate.permissions,
walkstate.memattrs.memtype,
ispriv, acctype, FALSE) then
        // The permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = FALSE;
    elseif AArch32.S2HasPermissionsFault(s2fslwalk, walkparams,
walkstate.permissions,
walkstate.memattrs.memtype,
ispriv, acctype, TRUE) then
        // The permission fault _was_ caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = TRUE;
    elseif AArch32.S2HasPermissionsFault(s2fslwalk, walkparams,
walkstate.permissions,

```

```

                                walkstate.memattrs.memtype,
                                ispriv, acctype, iswrite) then
    fault.statuscode = Fault\_Permission;

if ((s2fslwalk &&
    walkstate.memattrs.memtype == MemType\_Device) ||
    (acctype == AccType\_IFETCH &&
    (walkstate.memattrs.memtype == MemType\_Device || HCR2.ID == '1')) ||
    (acctype != AccType\_IFETCH &&
    walkstate.memattrs.memtype == MemType\_Normal && HCR2.CD == '1')) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_memattrs = NormalNCMemAttr();
    s2_memattrs.xs = walkstate.memattrs.xs;
else
    s2_memattrs = walkstate.memattrs;

memattrs = s2_memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
// Output Address
oa = StageOA(walkstate.baseaddress, ipa_64, walkparams.tgx, walkstate.level);
pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
(ipa.paddress.address<39:0>, 64);
// Output Address
oa = StageOA(walkstate.baseaddress, ipa_64, walkparams.tgx, walkstate.level);
pa = CreateAddressDescriptor(ipa.vaddress, oa, s2_memattrs);

return (fault, pa);

```

Library pseudocode for aarch32/translation/translation/AArch32.SDStageOA

```

// AArch32.SDStageOA()
// =====
// Given the final walk state of a short-descriptor translation walk,
// map the untranslated input address bits to the base output address

FullAddress AArch32.SDStageOA(FullAddress baseaddress, bits(32) va, SDType sdftype)
    case sdftype of
        when SDType\_SmallPage      tsize = 12;
        when SDType\_LargePage     tsize = 16;
        when SDType\_Section       tsize = 20;
        when SDType\_Supersection  tsize = 24;

    // Output Address
    FullAddress oa;
    oa.address = baseaddress.address<51:tsize>:va<tsize-1:0>;
    oa.paspace = baseaddress.paspace;
    return oa;

```

Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) va, AccType acctype,
                                           boolean iswrite, boolean aligned,
                                           integer size)

    regime = TranslationRegime(PSTATE.EL, acctype);
    if !RegimeUsingAArch32(regime) then
        return AArch64.TranslateAddress(ZeroExtend(va, 64), acctype, iswrite,
                                         aligned, size);
    result = AArch32.FullTranslate(va, acctype, iswrite, aligned);
    if !IsFault(result) then
        result.fault = AArch32.CheckDebug(va, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va);

    return result;
```

Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeLD

```
// AArch32.DecodeDescriptorTypeLD()
// =====
// Determine whether the long-descriptor is a page, block or table

DescriptorType AArch32.DecodeDescriptorTypeLD(bits(64) descriptor, integer level)
    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Page;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<1:0> == '01' && level != FINAL_LEVEL then
        return DescriptorType_Block;
    else
        return DescriptorType_Invalid;
```

Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeSD

```
// AArch32.DecodeDescriptorTypeSD()
// =====
// Determine the type of the short-descriptor

SDFType AArch32.DecodeDescriptorTypeSD(bits(32) descriptor, integer level)
    if level == 1 && descriptor<1:0> == '01' then
        return SDFType_Table;
    elsif level == 1 && descriptor<18,1> == '01' then
        return SDFType_Section;
    elsif level == 1 && descriptor<18,1> == '11' then
        return SDFType_Supersection;
    elsif level == 2 && descriptor<1:0> == '01' then
        return SDFType_LargePage;
    elsif level == 2 && descriptor<1:0> == '1x' then
        return SDFType_SmallPage;
    else
        return SDFType_Invalid;
```

Library pseudocode for aarch32/translation/walk/AArch32.S1IASize

```
// AArch32.S1IASize()
// =====
// Retrieve the number of bits containing the input address for stage 1 translation

integer AArch32.S1IASize(bits(3) txsz)
    return 32 - UInt(txsz);
```



```

// AArch32.S1WalkLD()
// =====
// Traverse stage 1 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState)(FaultRecord, TTWState) AArch32.S1WalkLD(FaultRecord fault, AArch32.S1WalkLD(FaultRecord fault, TTWState walkstate, bits(32) va)

    if regime == Regime_EL2 then
        ttbr = HTTBR;
        txsz = walkparams.t0sz;
    else
        assert TTBCR.EAE == '1';
        varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
        if varange == VARange_LOWER then
            ttbr = TTBR0;
            epd = TTBCR.EPD0;
            txsz = walkparams.t0sz;
        else
            ttbr = TTBR1;
            epd = TTBCR.EPD1;
            txsz = walkparams.t1sz;

    if regime != Regime_EL2 && epd == '1' then
        fault.level = 1;
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);

// Input Address size
iasize = TTWState UNKNOWN);

// Input Address size
iasize = AArch32.S1IASize(txsz);
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;
startlevel = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
levels = FINAL_LEVEL - startlevel;

if !IsZero(ttbr<47:40>) then
    fault.statuscode = Fault_AddressSize;
    fault.level = 0;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baselsb = iasize - (levels*stride + granulebits) + 3;
baseaddress.paspace = if TTWState UNKNOWN);

FullAddress baseaddress;
baselsb = iasize - (levels*stride + granulebits) + 3;
baseaddress.paspace = if IsSecure() then() then PAS_Secure else PAS_NonSecure;
baseaddress.address = PAS_Secure else PAS_NonSecure;
baseaddress.address = ZeroExtend(ttbr<39:baselsb>:Zeros(baselsb));(baselsb));

TTWState walkstate;
walkstate.baseaddress = baseaddress;
walkstate.level = startlevel;
walkstate.istable = TRUE;
walkstate.memattrs =

TTWState walkstate;
walkstate.baseaddress = baseaddress;
walkstate.level = startlevel;
walkstate.istable = TRUE;
walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
walkstate.permissions.ap_table = '00';
walkstate.permissions.xn_table = '0';
walkstate.permissions.pxn_table = '0';

indexmsb = iasize - 1;
bits(64) descriptor; bits(64) descriptor;
AddressDescriptor walkaddress;
repeat

```

```

    fault.level = walkstate.level;
    indexlsb = (
        AddressDescriptor walkaddress;
    repeat
        fault.level = walkstate.level;
        indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
        bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000');

        // VA is needed in the case of reporting an external abort
        walkaddress.vaddress = ZeroExtend(va);
        walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        disablecache = (if regime == Regime_EL2 then HSCTLR.C else SCTLR.C) == '0';
        if disablecache then
            walkaddress.memattrs = NormalNCMemAttr();
            walkaddress.memattrs.xs = walkstate.memattrs.xs;
        else
            walkaddress.memattrs = walkstate.memattrs;

        // Shareability of target memory subject to stage 2 translation
        // is maintained as input to stage 2.
        if (regime == Regime_EL1) // If the target memory for translation table walks is Normal memory
            // inner Non-Cacheable outer Non-Cacheable then it is forced to Outer Shareable
            // Note: this behaviour is overridden for addresses subject to stage 2.
            if (walkaddress.memattrs.inner.attrs == MemAttr_NC &&
                walkaddress.memattrs.outer.attrs == MemAttr_NC) then
                walkaddress.memattrs.shareability = Shareability_OSH;

        // If there are two stages of translation, then the first stage table walk addresses
        // are themselves subject to translation
        if regime == Regime_EL10 && EL2Enabled() &&
            (if() then
                // Shareability of target memory subject to stage 2 translation
                // is maintained as input to stage 2.
                if (if ELUsingAArch32(EL2) then HCR.VM else HCR_EL2.VM) == '1') then
                    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
            else
                walkaddress.memattrs.shareability = walkstate.memattrs.shareability then HCR.VM else HCR_EL2.VM) == '1' then
                    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;

        s2fslwalk = TRUE;
        s2acctype = NormaliseShareability(walkaddress.memattrs);

        // If there are two stages of translation, then the first stage table walk addresses
        // are themselves subject to translation
        if regime == Regime_EL10 && EL2Enabled() then
            s2fslwalk = TRUE;
            s2acctype = AccType_TTW;
            s2aligned = TRUE;
            s2write = FALSE;
            (s2fault, s2walkaddress) = (s2fault, s2walkaddress) = (s2fault, s2walkaddress) = AArch32.S2Translate(fault, v
                s2acctype, s2aligned, s2write);
            // Check for a fault on the stage 2 walk
            if s2fault.statuscode != AArch32.S2Translate(fault, walkaddress, s2fslwalk,
                s2acctype, s2aligned, s2write);
            // Check for a fault on the stage 2 walk
            if s2fault.statuscode != Fault_None then
                return (s2fault, TTWState UNKNOWN);

            (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault);
        else
            (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

        if fault.statuscode != TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

```

```

    if fault.statuscode != Fault_None then
        return (fault, return (fault, TTWState UNKNOWN););
desctype = TTWState UNKNOWN);
    desctype = AArch32.DecodeDescriptorTypeID(descriptor, walkstate.level);

    case desctype of
        when DescriptorType_Table
            if !IsZero(descriptor<47:40>) then
                fault.statuscode = Fault_AddressSize;
                return (fault, return (fault, TTWState UNKNOWN););
walkstate.baseaddress.address = TTWState UNKNOWN);
                walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12));
                if walkstate.baseaddress.paspace == if walkstate.baseaddress.paspace == PA
                    walkstate.baseaddress.paspace = PAS_NonSecure;

                if walkparams.hpd == '0' then
                    walkstate.permissions.xn_table = (walkstate.permissions.xn_table OR
                                                         descriptor<60>);
                    walkstate.permissions.ap_table = (walkstate.permissions.ap_table OR
                                                         descriptor<62:61>);
                    walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR
                                                         descriptor<59>);

                walkstate.level = walkstate.level + 1;
                indexmsb = indexlsb - 1;

            when PAS_Secure && descriptor<63> == '1' then
                walkstate.baseaddress.paspace = PAS_NonSecure;

                if walkparams.hpd == '0' then
                    walkstate.permissions.xn_table = (walkstate.permissions.xn_table OR
                                                         descriptor<60>);
                    walkstate.permissions.ap_table = (walkstate.permissions.ap_table OR
                                                         descriptor<62:61>);
                    walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR
                                                         descriptor<59>);

                walkstate.level = walkstate.level + 1;
                indexmsb = indexlsb - 1;

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, return (fault, TTWState UNKNOWN););
when TTWState UNKNOWN);
            when DescriptorType_Page, DescriptorType_Block
                walkstate.istable = FALSE;

    until desctype IN {DescriptorType_Page, DescriptorType_Block};

    // Check the output address is inside the supported range
    if !IsZero(descriptor<47:40>) then
        fault.statuscode = Fault_AddressSize;
        return (fault, return (fault, TTWState UNKNOWN););

    // Check the access flag
    if descriptor<10> == '0' then
        fault.statuscode = TTWState UNKNOWN);

    // Check the access flag
    if descriptor<10> == '0' then
        fault.statuscode = Fault_AccessFlag;
        return (fault, return (fault, TTWState UNKNOWN););

    walkstate.permissions.xn = descriptor<54>;

```

```

walkstate.permissions.pxn = descriptor<53>;
walkstate.permissions.ap  = descriptor<7:6>:'1';
walkstate.contiguous      = descriptor<52>;

walkstate.baseaddress.address = TTwState UNKNOWN);

walkstate.permissions.xn  = descriptor<54>;
walkstate.permissions.pxn = descriptor<53>;
walkstate.permissions.ap  = descriptor<7:6>:'1';
walkstate.contiguous      = descriptor<52>;

walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb));
if walkstate.baseaddress.paspace == PAS\_Secure && descriptor<5> == '1' then
    walkstate.baseaddress.paspace = PAS\_NonSecure;
(indexlsb));
if walkstate.baseaddress.paspace == PAS\_Secure && descriptor<5> == '1' then
    walkstate.baseaddress.paspace = PAS\_NonSecure;

memattr = descriptor<4:2>;
sh      = descriptor<9:8>;
attr    = MAIRAttr(UInt(memattr), walkparams.mair);
slaarch64 = FALSE;
walkstate.memattrs = S1DecodeMemAttrs(attr, sh, slaarch64);

return (fault, walkstate);

```



```

// AArch32.S1WalkSD()
// =====
// Traverse stage 1 translation tables in short format to obtain the final descriptor

(FaultRecord, TTWState)(FaultRecord, TTWState) AArch32.S1WalkSD(FaultRecord fault, AArch32.S1WalkSD(FaultRecord fault, TTWState TTWState))
    assert TTBCR.EAE == '0';

    // Determine correct Translation Table Base Register to use.
    n = UInt(TTBCR.N);
    if n == 0 || IsZero(va<31:(32-n)>) then
        ttb = TTBR0.TTB0:Zeros(7);
        pd = TTBCR.PD0;
        irgn = TTBR0.IRGN;
        rgn = TTBR0.RGN;
        s = TTBR0.S;
        nos = TTBR0.NOS;
    else
        n = 0; // TTBR1 translation always treats N as 0
        ttb = TTBR1.TTB1:Zeros(7);
        pd = TTBCR.PD1;
        irgn = TTBR1.IRGN;
        rgn = TTBR1.RGN;
        s = TTBR1.S;
        nos = TTBR1.NOS;

    // Check if Translation table walk disabled for translations with this Base register.
    if pd == '1' then
        fault.level = 1;
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baseaddress.paspace = if TTWState UNKNOWN;

    FullAddress baseaddress;
    baseaddress.paspace = if IsSecure() then() then PAS_Secure else PAS_NonSecure;
    baseaddress.address = PAS_Secure else PAS_NonSecure;
    baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n));(14-n));

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.memattrs =

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.memattrs = WalkMemAttrs(s:nos, irgn, rgn);
    walkstate.level = 1;
    walkstate.istable = TRUE;

    bits(4) domain;
    bits(32) descriptor; bits(32) descriptor;
    AddressDescriptor walkaddress;
    repeat
        fault.level = walkstate.level;

    bits(32) index;
    if walkstate.level == 1 then
        index =

    AddressDescriptor walkaddress;
    repeat
        fault.level = walkstate.level;

    bits(32) index;
    if walkstate.level == 1 then
        index = ZeroExtend(va<31-n:20>:'00');
    else
        index = ZeroExtend(va<19:12>:'00');

    walkaddress.vaddress = ZeroExtend(va);
    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);

```

```

walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

if SCTL.R.C == '0' then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability of target memory subject to stage 2 translation
// is maintained as input to stage 2.
if (regime == Regime_EL10 // If the target memory for translation table walks is Normal memory
    // inner Non-Cacheable outer Non-Cacheable then it is forced to Outer Shareable
    // Note: this behaviour is overridden for addresses subject to stage 2.
    if (walkaddress.memattrs.inner.attrs == MemAttr_NC &&
        walkaddress.memattrs.outer.attrs == MemAttr_NC) then
        walkaddress.memattrs.shareability = Shareability_OSH;

    if regime == Regime_EL10 && EL2Enabled() &&
        (if() then
            // Shareability of target memory subject to stage 2 translation
            // is maintained as input to stage 2.
            if (if ELUsingAArch32(EL2) then HCR.VM else HCR_EL2.VM) == '1') then
                walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
            else
                walkaddress.memattrs.shareability =) then HCR.VM else HCR_EL2.VM) == '1' then
                walkaddress.memattrs.shareability = walkstate.memattrs.shareability;

    s2fslwalk = TRUE;
    s2acctype = NormaliseShareability(walkaddress.memattrs);

    if regime == Regime_EL10 && EL2Enabled() then
        s2fslwalk = TRUE;
        s2acctype = AccType_TTW;
        s2aligned = TRUE;
        s2write = FALSE;
        (s2fault, s2walkaddress) = (s2fault, s2walkaddress) = AArch32.S2Translate(fault, v
            s2acctype, s2aligned, s2write);

        if s2fault.statuscode != AArch32.S2Translate(fault, walkaddress, s2fslwalk,
            s2acctype, s2aligned, s2write);

        if s2fault.statuscode != Fault_None then
            return (s2fault, return (s2fault, TTWState UNKNOWN));

        (fault, descriptor) = FetchDescriptor(SCTL.R.EE, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(SCTL.R.EE, walkaddress, fault);

    if fault.statuscode != TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(SCTL.R.EE, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(SCTL.R.EE, walkaddress, fault);

    if fault.statuscode != Fault_None then
        return (fault, return (fault, TTWState UNKNOWN));

    walkstate.sdftype = TTWState UNKNOWN);

    walkstate.sdftype = AArch32.DecodeDescriptorTypeSD(descriptor, walkstate.level);

    case walkstate.sdftype of
        when SDType_Invalid
            fault.domain = domain;
            fault.statuscode = Fault_Translation;
            return (fault, return (fault, TTWState UNKNOWN));

        when TTWState UNKNOWN);

        when SDType_Table

```

```

    domain = descriptor<8:5>;
    ns     = descriptor<3>;
    pxn    = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:10>:Zeros(10));
    walkstate.level = 2;

when SDFType_SmallPage
    nG = descriptor<11>;
    s  = descriptor<10>;
    ap = descriptor<9,5:4>;
    tex = descriptor<8:6>;
    c   = descriptor<3>;
    b   = descriptor<2>;
    xn  = descriptor<0>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:12>:Zeros(12));
    walkstate.istable = FALSE;

when SDFType_LargePage
    xn = descriptor<15>;
    tex = descriptor<14:12>;
    nG = descriptor<11>;
    s  = descriptor<10>;
    ap = descriptor<9,5:4>;
    c   = descriptor<3>;
    b   = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:16>:Zeros(16));
    walkstate.istable = FALSE;

when SDFType_Section
    ns     = descriptor<19>;
    nG     = descriptor<17>;
    s      = descriptor<16>;
    ap     = descriptor<15,11:10>;
    tex    = descriptor<14:12>;
    domain = descriptor<8:5>;
    xn     = descriptor<4>;
    c      = descriptor<3>;
    b      = descriptor<2>;
    pxn    = descriptor<0>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:20>:Zeros(20));
    walkstate.istable = FALSE;

when SDFType_Supersection
    ns     = descriptor<19>;
    nG     = descriptor<17>;
    s      = descriptor<16>;
    ap     = descriptor<15,11:10>;
    tex    = descriptor<14:12>;
    xn     = descriptor<4>;
    c      = descriptor<3>;
    b      = descriptor<2>;
    pxn    = descriptor<0>;
    domain = '0000';

    walkstate.baseaddress.address = ZeroExtend(descriptor<8:5,23:20,31:24>:Zeros(24));
    walkstate.istable = FALSE;

until walkstate.sdftype != SDFType_Table;

if SCTL.AFE == '1' && ap<0> == '0' then
    fault.domain      = domain;
    fault.statuscode = Fault_AccessFlag;
    return (fault, return (fault, TTWState UNKNOWN););

// Decode the TEX, C, B and S bits to produce target memory attributes
if SCTL.TRE == '1' then

```

```

walkstate.memattrs = TTWState UNKNOWN);

// Decode the TEX, C, B and S bits to produce target memory attributes
if SCTL.TRE == '1' then
    walkstate.memattrs = AArch32.RemappedTEXDecode(tex, c, b, s);
elsif RemapRegsHaveResetValues() then
    walkstate.memattrs = AArch32.DefaultTEXDecode(tex, c, b, s);
else
    walkstate.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;

walkstate.permissions.ap = ap;
walkstate.permissions.xn = xn;
walkstate.permissions.pxn = pxn;
walkstate.domain = domain;

if IsSecure() && ns == '0' then
    walkstate.baseaddress.paspace = PAS_Secure;
else
    walkstate.baseaddress.paspace = PAS_NonSecure;
() && ns == '0' then
    walkstate.baseaddress.paspace = PAS_Secure;
else
    walkstate.baseaddress.paspace = PAS_NonSecure;

return (fault, walkstate);

```

Library pseudocode for aarch32/translation/walk/AArch32.S2IASize

```

// AArch32.S2IASize()
// =====
// Retrieve the number of bits containing the input address for stage 2 translation

integer AArch32.S2IASize(bits(4) t0sz)
    return 32 - SInt(t0sz);

```

Library pseudocode for aarch32/translation/walk/AArch32.S2StartLevel

```

// AArch32.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch32.S2StartLevel(bits(2) sl0)
    return 2 - UInt(sl0);

```



```

// AArch32.S2Walk()
// =====
// Traverse stage 2 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState)(FaultRecord, TTWState) AArch32.S2Walk(FaultRecord fault, AArch32.S2Walk(FaultRecord
AddressDescriptor ipa)

if walkparams.sl0 == '1x' ||
AddressDescriptor ipa)

if walkparams.sl0 == '1x' || AArch32.S2InconsistentSL(walkparams) then
    fault.statuscode = Fault_Translation;
    fault.level = 1;
    return (fault, return (fault, TTWState UNKNOWN);

// Input Address size
iasize = TTWState UNKNOWN);

// Input Address size
iasize = AArch32.S2IASize(walkparams.t0sz);
startlevel = AArch32.S2StartLevel(walkparams.sl0);
levels = FINAL_LEVEL - startlevel;
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;

if !IsZero(VTTBR<47:40>) then
    fault.statuscode = Fault_AddressSize;
    fault.level = 0;
    return (fault, return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baselsb = iasize - (levels*stride + granulebits) + 3;
baseaddress.paspace = PAS_NonSecure;
baseaddress.address = TTWState UNKNOWN);

FullAddress baseaddress;
baselsb = iasize - (levels*stride + granulebits) + 3;
baseaddress.paspace = PAS_NonSecure;
baseaddress.address = ZeroExtend(VTTBR<39:baselsb>:Zeros(baselsb));(baselsb));

TTWState walkstate;
walkstate.baseaddress = baseaddress;
walkstate.level = startlevel;
walkstate.istable = TRUE;
walkstate.memattrs =

TTWState walkstate;
walkstate.baseaddress = baseaddress;
walkstate.level = startlevel;
walkstate.istable = TRUE;
walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn,
walkparams.orgn);

indexmsb = iasize - 1;
bits(64) descriptor; bits(64) descriptor;
AddressDescriptor walkaddress;
repeat
    fault.level = walkstate.level;

    indexlsb = (
AddressDescriptor walkaddress;
repeat
    fault.level = walkstate.level;

indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
bits(40) index = ZeroExtend(ipa.address.address<indexmsb:indexlsb>:'000');

// Update virtual address for abort functions
walkaddress.vaddress = ipa.vaddress;
walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);

```

```

walkaddress.paddress.paspace = walkstate.baseaddress.paspace;
if HCR2.CD == '1' then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

walkaddress.memattrs.shareability = _____ // If the target memory for translation table walk
// inner Non-Cacheable outer Non-Cacheable then it is forced to Outer Shareable
if (walkaddress.memattrs.inner.attrs == NormaliseShareabilityMemAttr_NC(walkaddress.memattrs,
(fault, descriptor) = ==
walkaddress.memattrs.outer.attrs == FetchDescriptorMemAttr_NC(walkparams.ee, walkaddr

if fault.statuscode !=) then
    walkaddress.memattrs.shareability = Shareability_OSH;

(fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

if fault.statuscode != Fault_None then
    return (fault, _____ return (fault, TTWState UNKNOWN);

desctype = TTWState UNKNOWN);

desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

case desctype of
    when DescriptorType_Table
        if !IsZero(descriptor<47:40>) then
            fault.statuscode = Fault_AddressSize;
            return (fault, _____ return (fault, TTWState UNKNOWN);

            walkstate.baseaddress.address = TTWState UNKNOWN);

            walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12));
            walkstate.level = walkstate.level + 1;
            indexmsb = indexlsb - 1;

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, _____ return (fault, TTWState UNKNOWN);

                when TTWState UNKNOWN);

            when DescriptorType_Page, DescriptorType_Block
                walkstate.istable = FALSE;

until desctype IN {DescriptorType_Page, DescriptorType_Block};

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault_AddressSize;
    return (fault, _____ return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault_AccessFlag;
    return (fault, TTWState UNKNOWN);
;
return (fault, TTWState UNKNOWN);

// Unpack the descriptor into address and upper and lower block attributes
walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb));

walkstate.permissions.s2ap = descriptor<7:6>;
walkstate.permissions.s2xn = descriptor<54>;

```



```

if HaveExtendedExecuteNeverExt() then
    walkstate.permissions.s2xnx = descriptor<53>;
else
    walkstate.permissions.s2xnx = '0';

memattr = descriptor<5:2>;
sh       = descriptor<9:8>;
walkstate.memattrs  = S2DecodeMemAttrs(memattr, sh);
walkstate.contiguous = descriptor<52>;

return (fault, walkstate);

```

Library pseudocode for aarch32/translation/walk/AArch32.TranslationSizeSD

```

// AArch32.TranslationSizeSD()
// =====
// Determine the size of the translation

integer AArch32.TranslationSizeSD(SDFTYPE sdftype)
    case sdftype of
        when SDFTYPE_SmallPage      tsize = 12;
        when SDFTYPE_LargePage      tsize = 16;
        when SDFTYPE_Section        tsize = 20;
        when SDFTYPE_Supersection   tsize = 24;

    return tsize;

```

Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

```

boolean RemapRegsHaveResetValues();

```

Library pseudocode for aarch32/translation/walkparams/AArch32.GetS1TTWParams

```

// AArch32.GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective controlling
// system registers.

S1TTWParams AArch32.GetS1TTWParams(Regime regime, bits(32) va)
    S1TTWParams walkparams;

    case regime of
        when Regime_EL2    walkparams = AArch32.S1TTWParamsPL2();
        when Regime_EL10   walkparams = AArch32.S1TTWParamsPL10(va);
        when Regime_EL30   walkparams = AArch32.S1TTWParamsPL10(va);

    return walkparams;

```

Library pseudocode for aarch32/translation/walkparams/AArch32.GetS2TTWParams

```
// AArch32.GetS2TTWParams()
// =====
// Gather walk parameters for stage 2 translation

S2TTWParams AArch32.GetS2TTWParams()
    S2TTWParams walkparams;

    walkparams.tgx = TGx_4KB;
    walkparams.s = VTCR.S;
    walkparams.t0sz = VTCR.T0SZ;
    walkparams.sl0 = VTCR.SL0;
    walkparams.irgn = VTCR.IRGN0;
    walkparams.orgn = VTCR.ORGNO;
    walkparams.sh = VTCR.SH0;
    walkparams.ee = HSCTLR.EE;
    walkparams.ptw = HCR.PTW;
    walkparams.vm = HCR.VM OR HCR.DC;

    // VTCR.S must match VTCR.T0SZ[3]
    if walkparams.s != walkparams.t0sz<3> then
        (-, walkparams.t0sz) = ConstrainUnpredictableBits(Unpredictable_RESVTCRS);

    return walkparams;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.GetVARange

```
// AArch32.GetVARange()
// =====
// Select the translation base address for stage 1 long-descriptor walks

VARange AArch32.GetVARange(bits(32) va, bits(3) t0sz, bits(3) t1sz)
    // Lower range Input Address size
    lo_iasize = AArch32.S1IASize(t0sz);
    // Upper range Input Address size
    up_iasize = AArch32.S1IASize(t1sz);

    if t1sz == '000' && t0sz == '000' then
        return VARange_LOWER;
    elsif t1sz == '000' then
        return if IsZero(va<31:lo_iasize>) then VARange_LOWER else VARange_UPPER;
    elsif t0sz == '000' then
        return if IsOnes(va<31:up_iasize>) then VARange_UPPER else VARange_LOWER;
    elsif IsZero(va<31:lo_iasize>) then
        return VARange_LOWER;
    elsif IsOnes(va<31:up_iasize>) then
        return VARange_UPPER;
    else
        // Will be reported as a Translation Fault
        return VARange UNKNOWN;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL2

```
// AArch32.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch32.S1TTWParamsPL2()
    S1TTWParams walkparams;

    walkparams.tgx = TGx_4KB;
    walkparams.t0sz = HTCR.T0SZ;
    walkparams.irgn = HTCR.SH0;
    walkparams.orgn = HTCR.IRGN0;
    walkparams.sh = HTCR.ORGNO;
    walkparams.hpd = if AArch32.HaveHPDExt() then HTCR.HPD else '0';
    walkparams.ee = HSCTLR.EE;
    walkparams.wxn = HSCTLR.WXN;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmid = HSCTLR.nTLSMD;
    else
        walkparams.ntlsmid = '1';

    walkparams.mair = HMAIR1:HMAIR0;

    return walkparams;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsPL10

```
// AArch32.S1TTWParamsPL10()
// =====
// Gather stage 1 translation table walk parameters for EL3&0 regime as well as
// EL1&0 regime (with EL2 enabled or disabled)

S1TTWParams AArch32.S1TTWParamsPL10(bits(32) va)
    assert TTBCR.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz = TTBCR.T0SZ;
    walkparams.t1sz = TTBCR.T1SZ;
    walkparams.ee = SCTLR.EE;
    walkparams.wxn = SCTLR.WXN;
    walkparams.uwxn = SCTLR.UWXN;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmid = SCTLR.nTLSMD;
    else
        walkparams.ntlsmid = '1';

    walkparams.mair = MAIR1:MAIR0;
    walkparams.sif = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh = TTBCR.SH0;
        walkparams.irgn = TTBCR.IRGN0;
        walkparams.orgn = TTBCR.ORGNO;
        if AArch32.HaveHPDExt() then
            walkparams.hpd = TTBCR.T2E AND TTBCR2.HPD0;
        else
            walkparams.hpd = '0';
    else
        walkparams.sh = TTBCR.SH1;
        walkparams.irgn = TTBCR.IRGN1;
        walkparams.orgn = TTBCR.ORGNO;
        if AArch32.HaveHPDExt() then
            walkparams.hpd = TTBCR.T2E AND TTBCR2.HPD1;
        else
            walkparams.hpd = '0';

    return walkparams;
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =====
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccType acctype, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplementedGetNumBreakpoints();

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                     linked, DBGBCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAArch32HaveAnyAArch32() && size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    match = value_match && state_match && enabled;

    return match;
```



```

// AArch64.BreakpointValueMatch()
// =====

boolean AArch64.BreakpointValueMatch(integer n, bits(64) vaddress, boolean linked_to)

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n >= NumBreakpointsImplementedGetNumBreakpoints() then
    (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplementedGetNumBreakpoints() - 1, Unpre
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return FALSE;

// If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
// call from StateMatch for linking).
if DBGBCR_EL1[n].E == '0' then return FALSE;

context_aware = (n >= (NumBreakpointsImplementedGetNumBreakpoints() - NumContextAwareBreakpointsImple

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
dbgtype = DBGBCR_EL1[n].BT;

if ((dbgtype IN {'011x', '11xx'}) && !HaveVirtHostExt() && !HaveV82Debug()) || // Context matching
    dbgtype == '010x' || // Reserved
    (dbgtype != '0x0x' && !context_aware) || // Context matching
    (dbgtype == '1xxx' && !HaveEL(EL2))) then // EL2 extension
    (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (dbgtype == '0x0x');
match_vmid = (dbgtype == '10xx');
match_cid = (dbgtype == '001x');
match_cid1 = (dbgtype IN {'101x', 'x11x'});
match_cid2 = (dbgtype == '11xx');
linked = (dbgtype == 'xxx1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, or if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return FALSE;

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return FALSE;

// Do the comparison.
if match_addr then
    byte = UInt(vaddress<1:0>);
    if HaveAArch32HaveAnyAArch32() then
        // T32 instructions can be executed at EL0 in an AArch64 translation regime.
        assert byte IN {0,2}; // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
    else
        assert byte == 0; // "vaddress" is word aligned
        byte_select_match = TRUE; // DBGBCR_EL1[n].BAS<byte> is RES1
    // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
    // of DBGxVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
    // included in the match.
    // If 'vaddress' is outside of the current virtual address space, then the access
    // generates a Translation fault.
    integer top = AArch64.VAMax();
    if !IsOnes(DBGxVR_EL1[n]<63:top>) && !IsZero(DBGxVR_EL1[n]<63:top>) then
        if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
            top = 63;

```

```

    BVR_match = (vaddress<top:2> == DBGBVR_EL1[n]<top:2>) && byte_select_match;

elseif match_cid then
    if IsInHost\(\) then
        BVR_match = (CONTEXTIDR_EL2<31:0> == DBGBVR_EL1[n]<31:0>);
    else
        BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);
elseif match_cid1 then
    BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost\(\) && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);
if match_vmid then
    if !Have16bitVMID\(\) || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGBVR_EL1[n]<47:32>;
    BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) &&
        !IsInHost\(\) &&
        vmid == bvr_vmid);
elseif match_cid2 then
    BXVR_match = (PSTATE.EL != EL3 && ({} &&
        vmid == bvr_vmid));
elseif match_cid2 then
    BXVR_match = ((HaveVirtHostExt\(\) || HaveV82Debug\(\)) &&
        EL2Enabled\(\) &&
        DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);

bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

return match;

```

Library pseudocode for aarch64/debug/breakpoint/AArch64.StateMatch

```
// AArch64.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreaknt, AccType acctype, boolean ispriv)
// "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreaknt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

// If parameters are set to a reserved type, behaves as either disabled or a defined type
(c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreaknt);
if c == Constraint\_DISABLED then return FALSE;
// Otherwise the HMC,SSC,PxC values are either valid or the values returned by
// CheckValidStateMatch are valid.

EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
EL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
EL1_match = PxC<0> == '1';
EL0_match = PxC<1> == '1';

if HaveNV2Ext() && acctype == AccType\_NV2REGISTER && !isbreaknt then
    priv_match = EL2_match;
elsif !ispriv && !isbreaknt then
    priv_match = EL0_match;
else
    case PSTATE.EL of
        when EL3 priv_match = EL3_match;
        when EL2 priv_match = EL2_match;
        when EL1 priv_match = EL1_match;
        when EL0 priv_match = EL0_match;

case SSC of
    when '00' security_state_match = TRUE; // Both
    when '01' security_state_match = !IsSecure(); // Non-secure only
    when '10' security_state_match = IsSecure(); // Secure only
    when '11' security_state_match = (HMC == '1' || IsSecure()); // HMC=1 -> Both, 0 -> Secure only

if linked then
    // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
    // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    lbn = UInt(LBN);
    first_ctx_cmp = NumBreakpointsImplementedGetNumBreakpoints() - NumContextAwareBreakpointsImplemented
    last_ctx_cmp = NumBreakpointsImplementedGetNumBreakpoints() - 1;
    if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
        (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable\_BPN0TCTX0);
        assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNKNOWN};
        case c of
            when Constraint\_DISABLED return FALSE; // Disabled
            when Constraint\_NONE linked = FALSE; // No linking
            // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

if linked then
    vaddress = bits(64) UNKNOWN;
    linked_to = TRUE;
    linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

return priv_match && security_state_match && (!linked || linked_match);
```


Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =====

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);
```

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = HaveEL(EL2) && (!secure || IsSecureEL2Enabled()) && (HCR_EL2.TGE == '1' || MDCR_EL2.TD0 == '1');
    target = (if route_to_el2 then EL2 else EL1);
    if enabled = ! HaveEL(EL3) && secure then
        enabled = MDCR_EL3.SDD == '0';
        if from == target then
            enabled = enabled && MDCR_EL1.KDE == '1' && mask == '0';
        else
            enabled = enabled && EL0 && ELUsingAArch32(EL1) then
                enabled = enabled || SDER32_EL3.SUIDEN == '1';
            else
                enabled = TRUE;

    if from == target then
        enabled = enabled && MDCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from);

    return enabled;
```

Library pseudocode for aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```
// AArch64.CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch64.CheckForPMUOverflow()

    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSSET_EL0<31> == '1';
    for n = 0 to NumEventCountersImplementedGetNumEventCounters() - 1
        if HaveEL(EL2) then
            E = (if n < UInt(MDCR_EL2.HPMN) then PMCR_EL0.E else MDCR_EL2.HPME);
        else
            E = PMCR_EL0.E;
        if E == '1' && PMINTENSET_EL1<n> == '1' && PMOVSSET_EL0<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

    return pmuirq;
```



```

// AArch64.CountEvents()
// =====
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch64.CountEvents(integer n)
    assert n == 31 || n < NumEventCountersImplementedGetNumEventCounters();

    // Event counting is disabled in Debug state
    debug = Halted();

    // In Non-secure state, some counters are reserved for EL2
    if HaveEL(EL2) then
        resvd_for_el2 = n >= UInt(MDCR_EL2.HPMN) && n != 31;
    else
        resvd_for_el2 = FALSE;

    // Main enable controls
    E = if resvd_for_el2 then MDCR_EL2.HPME else PMCR_EL0.E;
    enabled = E == '1' && PMCNTENSET_EL0<n> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;
    // Event counting in Secure state is prohibited if all of:
    // * EL3 is implemented
    // * MDCR_EL3.SPME == 0, and either:
    //   - FEAT_PMUv3p7 is not implemented
    //   - MDCR_EL3.MPMX == 0
    if HaveEL(EL3) && IsSecure() then
        if HavePMUv3p7() then
            prohibited = MDCR_EL3.<SPME,MPMX> == '00';
        else
            prohibited = MDCR_EL3.SPME == '0';

    // Event counting at EL3 is prohibited if all of:
    // * FEAT_PMUv3p7 is implemented
    // * One of the following is true:
    //   - MDCR_EL3.SPME == 0
    //   - MDCR_EL3.SPME == 1
    //   - PMNx is not reserved for EL2
    // * MDCR_EL3.MPMX == 1
    if !prohibited && PSTATE.EL == EL3 && HavePMUv3p7() then
        prohibited = MDCR_EL3.MPMX == '1' && (MDCR_EL3.SPME == '0' || !resvd_for_el2);
        prohibited = MDCR_EL3.MPMX == '1' && (MDCR_EL3.SPME == '1' || !resvd_for_el2);

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * PMNx is not reserved for EL2
    // * MDCR_EL2.HPMD == 1
    if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
        prohibited = MDCR_EL2.HPMD == '1';

    // The IMPLEMENTATION DEFINED authentication interface might override software
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // PMCR_EL0.DP disables the cycle counter when event counting is prohibited
    if enabled && prohibited && n == 31 then
        enabled = PMCR_EL0.DP == '0';

    // If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
    // This is not overridden by PMCR_EL0.DP.
    if Havev85PMU() && n == 31 then
        if HaveEL(EL3) && IsSecure() && MDCR_EL3.SCCD == '1' then
            prohibited = TRUE;
        if PSTATE.EL == EL2 && MDCR_EL2.HCCD == '1' then
            prohibited = TRUE;

    // If FEAT_PMUv3p7 is implemented, cycle counting can be prohibited at EL3.
    // This is not overridden by PMCR_EL0.DP.
    if HavePMUv3p7() && n == 31 then

```

```

    if PSTATE.EL == EL3 && MDCR_EL3.MCCD == '1' then
        prohibited = TRUE;

// Event counting might be frozen
frozen = FALSE;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() && n != 31 then
    ovflw = PMOVSLR_EL0<NumEventCountersImplementedGetNumEventCounters()-1:0>;
    if resvd_for_el2 then
        FZ = MDCR_EL2.HPMFZ0;
        ovflw<UInt(MDCR_EL2.HPMN)-1:0> = Zeros();
    else
        FZ = PMCR_EL0.FZ0;
        if HaveEL(EL2) then
            ovflw<NumEventCountersImplementedGetNumEventCounters()-1:UInt(MDCR_EL2.HPMN)> = Zeros();
        frozen = FZ == '1' && !IsZero(ovflw);

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH} bits
filter = if n == 31 then PMCCFILTR_EL0<31:0> else PMEVTYPER_EL0[n]<31:0>;

P   = filter<31>;
U   = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';
M   = if HaveEL(EL3) then filter<26> else '0';
SH  = if HaveEL(EL3) && HaveSecureEL2Ext() then filter<24> else '0';

case PSTATE.EL of
    when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
    when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
    when EL2 filtered = if IsSecure() then NSH == SH else NSH == '0';
    when EL3 filtered = M != P;

return !debug && enabled && !prohibited && !filtered && !frozen;

```

Library pseudocode for aarch64/debug/statisticalprofiling/CheckProfilingBufferAccess

```

// CheckProfilingBufferAccess()
// =====

SysRegAccess CheckProfilingBufferAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;

    if PSTATE.EL == EL1 && EL2Enabled() && MDCR_EL2.E2PB<0> != '1' then
        return SysRegAccess_TrapToEL2;

    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;

    return SysRegAccess_OK;

```

Library pseudocode for aarch64/debug/statisticalprofiling/CheckStatisticalProfilingAccess

```
// CheckStatisticalProfilingAccess()
// =====

SysRegAccess CheckStatisticalProfilingAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;

    if PSTATE.EL == EL1 && EL2Enabled() && MDCR_EL2.TPMS == '1' then
        return SysRegAccess_TrapToEL2;

    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;

    return SysRegAccess_OK;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// =====

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```
// CollectContextIDR2()
// =====

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if !EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// =====

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    if ((!secure && HaveEL(EL2)) || IsSecureEL2Enabled()) then
        return PMSCR_EL2.PA == '1' && (el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// =====

TimeStamp CollectTimeStamp()

    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (-, el) = ProfilingBufferOwner();

    if el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp_None;

    if !HaveECVExt() then
        PCT_el1 = '0':PMSCR_EL1.PCT<0>;          // PCT<1> is RES0
    else
        PCT_el1 = PMSCR_EL1.PCT;
        if PCT_el1 == '10' then
            // Reserved value
            (-, PCT_el1) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT);
    if EL2Enabled() then
        if !HaveECVExt() then
            PCT_el2 = '0':PMSCR_EL2.PCT<0>;      // PCT<1> is RES0
        else
            PCT_el2 = PMSCR_EL2.PCT;
            if PCT_el2 == '10' then
                // Reserved value
                (-, PCT_el2) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT);
    case PCT_el2 of
        when '00'
            return TimeStamp_Virtual;
        when '01'
            if el == EL2 then return TimeStamp_Physical;
        when '11'
            assert HaveECVExt();                  // FEAT_ECV must be implemented
            if el == EL1 && PCT_el1 == '00' then
                return TimeStamp_Virtual;
            else
                return TimeStamp_OffsetPhysical;
        otherwise
            Unreachable();

    case PCT_el1 of
        when '00' return TimeStamp_Virtual;
        when '01' return TimeStamp_Physical;
        when '11'
            assert HaveECVExt();                  // FEAT_ECV must be implemented
            return TimeStamp_OffsetPhysical;
        otherwise Unreachable();
```

Library pseudocode for aarch64/debug/statisticalprofiling/OpType

```
enumeration OpType {
    OpType_Load,          // Any memory-read operation other than atomics, compare-and-swap, and swap
    OpType_Store,         // Any memory-write operation, including atomics without return
    OpType_LoadAtomic,    // Atomics with return, compare-and-swap and swap
    OpType_Branch,        // Software write to the PC
    OpType_Other          // Any other class of operation
};
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// =====

boolean ProfilingBufferEnabled()
    if !HaveStatisticalProfiling() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    non_secure_bit = if secure then '0' else '1';
    return (!ELUsingArch32(el) && non_secure_bit == SCR_EL3.NS &&
        PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// =====

(boolean, bits(2)) ProfilingBufferOwner()
    secure = if HaveEL(EL3) then (MDCR_EL3.NSPB<1> == '0') else IsSecure();
    el = if HaveEL(EL2) && (!secure || IsSecureEL2Enabled()) && MDCR_EL2.E2PB == '00' then EL2 else EL1;
    return (secure, el);
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPECollectRecord

```
// SPECollectRecord()
// =====
// Returns TRUE if the sampled class of instructions or operations, as
// determined by PMSFCR_EL1, are recorded and FALSE otherwise.

boolean SPECollectRecord(bits(64) events, integer total_latency, OpType optype)
    assert StatisticalProfilingEnabled();

    bits(64) mask = 0xAA<63:0>; // Bits [7,5,3,1]
    if HaveSVE() then mask<18:17> = Ones(); // Predicate flags
    if HaveStatisticalProfilingv1p1() then mask<11> = '1'; // Alignment Flag
    if HaveStatisticalProfilingv1p2() then mask<6> = '1'; // Not taken flag
    mask<63:48> = bits(16) IMPLEMENTATION_DEFINED;
    mask<31:24> = bits(8) IMPLEMENTATION_DEFINED;
    mask<15:12> = bits(4) IMPLEMENTATION_DEFINED;

    // Check for UNPREDICTABLE case
    if (HaveStatisticalProfilingv1p2() && PMSFCR_EL1.<FnE,FE> == '11' &&
        !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
        if ConstrainUnpredictableBool(Unpredictable_BADPMSFCR) then
            return FALSE;
    else
        // Filtering by event
        if PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1) then
            e = events AND mask;
            m = PMSEVFR_EL1 AND mask;
            if !IsZero(NOT(e) AND m) then return FALSE;

        // Filtering by inverse event
        if (HaveStatisticalProfilingv1p2() && PMSFCR_EL1.FnE == '1' &&
            !IsZero(PMSNEVFR_EL1)) then
            e = events AND mask;
            m = PMSNEVFR_EL1 AND mask;
            if !IsZero(e AND m) then return FALSE;

    // Filtering by type
    if PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>) then
        case optype of
            when OpType_Branch
                if PMSFCR_EL1.B == '0' then return FALSE;
            when OpType_Load
                if PMSFCR_EL1.LD == '0' then return FALSE;
            when OpType_Store
                if PMSFCR_EL1.ST == '0' then return FALSE;
            when OpType_LoadAtomic
                if PMSFCR_EL1.<LD,ST> == '00' then return FALSE;
            otherwise
                return FALSE;

    // Filtering by latency
    if PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT) then
        if total_latency < UInt(PMSLATFR_EL1.MINLAT) then
            return FALSE;

    // Check for UNPREDICTABLE cases
    if ((PMSFCR_EL1.FE == '1' && IsZero(PMSEVFR_EL1 AND mask)) ||
        (PMSFCR_EL1.FT == '1' && IsZero(PMSFCR_EL1.<B,LD,ST>)) ||
        (PMSFCR_EL1.FL == '1' && IsZero(PMSLATFR_EL1.MINLAT))) then
        return ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    if (HaveStatisticalProfilingv1p2() &&
        ((PMSFCR_EL1.FnE == '1' && IsZero(PMSNEVFR_EL1 AND mask)) ||
        (PMSFCR_EL1.<FnE,FE> == '11' &&
            !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)))) then
        return ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    return TRUE;
```


Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =====

boolean StatisticalProfilingEnabled()
    if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    in_host = EL2Enabled() && HCR_EL2.TGE == '1';
    (secure, el) = ProfilingBufferOwner();
    if UInt(el) < UInt(PSTATE.EL) || secure != IsSecure() || (in_host && el == EL1) then
        return FALSE;

    case PSTATE.EL of
        when EL3 Unreachable();
        when EL2 spe_bit = PMSCR_EL2.E2SPE;
        when EL1 spe_bit = PMSCR_EL1.E1SPE;
        when EL0 spe_bit = (if in_host then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);

    return spe_bit == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/SysRegAccess

```
enumeration SysRegAccess { SysRegAccess_OK,
                           SysRegAccess_UNDEFINED,
                           SysRegAccess_TrapToEL1,
                           SysRegAccess_TrapToEL2,
                           SysRegAccess_TrapToEL3 };
```

Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```
enumeration TimeStamp {
    TimeStamp_None,           // No timestamp
    TimeStamp_CoreSight,      // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,        // Physical counter value with no offset
    TimeStamp_OffsetPhysical,  // Physical counter value minus CNTPOFF_EL2
    TimeStamp_Virtual };      // Physical counter value minus CNTVOFF_EL2
```

Library pseudocode for aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// =====
// Take an exception in Debug state to an Exception level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    if HaveIESB() then
        sync_errors = SCTRL[target_el].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        // SCTRL[].IESB and/or SCR_EL3.NMEA (if applicable) might be ignored in Debug state.
        if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
            sync_errors = FALSE;
    else
        sync_errors = FALSE;

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(target_el);

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = bits(64) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0'; // PSTATE.J is RES0
    if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTRL[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUA0Ext() then PSTATE.UA0 = '0';
    if HaveBTIExt() then PSTATE.BTYPE = '00';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    if HaveMTEEExt() then PSTATE.TCO = '1';

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    if sync_errors then
        SynchronizeErrors();

    EndOfInstruction();
```

Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// =====

boolean AArch64.WatchpointByteMatch(integer n, AccType acctype, bits(64) vaddress)

integer top = AArch64.VAMax();
bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;           // Word or doubleword
byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
mask = UInt(DBGWCR_EL1[n].MASK);

// If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
// DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
// UNPREDICTABLE.
if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
    byte_select_match = ConstrainUnpredictableBool(Unpredictable\_WPMASKANDBAS);
else
    LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1)); MSB = (DBGWCR_EL1[n].BAS + LSB);
    if !IsZero(MSB AND (MSB - 1)) then                        // Not contiguous
        byte_select_match = ConstrainUnpredictableBool(Unpredictable\_WPBASCONTIGUOUS);
        bottom = 3;                                          // For the whole doubleword

// If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask > 0 && mask <= 2 then
    (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable\_RESWPMASK);
    assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNKNOWN};
    case c of
        when Constraint\_DISABLED return FALSE;           // Disabled
        when Constraint\_NONE mask = 0;                   // No masking
        // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

if mask > bottom then
    // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
    // of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
    // included in the match.
    if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
        if ConstrainUnpredictableBool(Unpredictable\_DBGxVR\_RESS) then
            top = 63;
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable\_WPMASKEDBITS);
    else
        WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

return WVR_match && byte_select_match;
```

Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =====
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
                                AccType acctype, boolean iswrite)
assert !ELUsingAArch32(S1TranslationRegime());
assert n < NumWatchpointsImplementedGetNumWatchpoints();

// "ispriv" is:
// * FALSE for all loads, stores, and atomic operations executed at EL0.
// * FALSE if the access is unprivileged.
// * TRUE for all other loads, stores, and atomic operations.

enabled = DBGWCR_EL1[n].E == '1';
linked = DBGWCR_EL1[n].WT == '1';
isbreakpnt = FALSE;

state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
ls_match = FALSE;
if acctype == AccType_ATOMICRW then
    ls_match = (DBGWCR_EL1[n].LSC != '00');
else
    ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');

value_match = FALSE;
for byte = 0 to size - 1
    value_match = value_match || AArch64.WatchpointByteMatch(n, acctype, vaddress + byte);

return value_match && state_match && ls_match && enabled;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// =====
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

if IsDebugException(fault) then
    if fault.acctype == AccType_IFETCH then
        if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
            AArch64.VectorCatchException(fault);
        else
            AArch64.BreakpointException(fault);
    else
        AArch64.WatchpointException(vaddress, fault);
elseif fault.acctype == AccType_IFETCH then
    AArch64.InstructionAbort(vaddress, fault);
else
    AArch64.DataAbort(vaddress, fault);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(64) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint, Exception_M
    (exception.syndrome, exception.syndrome2) = AArch64.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        exception.ipaddress = fault.ipaddress.address;
    else
        exception.ipavalid = FALSE;

    return exception;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// =====

AArch64.CheckPCAlignment()

    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// =====
AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 =
AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = ( route_to_el2 = (PSTATE.EL IN {EL2Enabled()} && PSTATE.EL IN {EL0, EL1} &&
        (HCR_EL2.TGE == '1' ||
        ) && EL2Enabled() && (HCR_EL2.TGE == '1' ||
        (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
        (HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER) || ) ||
        IsSecondStage(fault)));
    bits(64) preferred_exception_return =
        IsSecondStage(fault));
    bits(64) preferred_exception_return = ThisInstrAddr();
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
        IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = exception = AArch64.AbortSyndrome( AArch64.AbortSyndrome(Exception_NV2DataAbort,
    else
        exception = exception = AArch64.AbortSyndrome( AArch64.AbortSyndrome(Exception_DataAbort,
        bits(2) target_el = if PSTATE.EL == EL1;
        if PSTATE.EL == EL3 || route_to_el3 then
            target_el = || route_to_el3 then AArch64.TakeException(EL3;
        , exception, preferred_exception_return, vect_offset);
        elsif PSTATE.EL == EL2 || route_to_el2 then
            target_el = || route_to_el2 then AArch64.TakeException(EL2; , exception, preferred_exception_return
        else
            AArch64.TakeException(EL1(target_el, exception, preferred_exception_return, vect_offset); , exception,
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.EffectiveTCF

```
// AArch64.EffectiveTCF()
// =====
// Returns the TCF field applied to tag check faults in the given Exception level.

bits(2) AArch64.EffectiveTCF(AccType acctype)
    bits(2) tcf, el;
    el = S1TranslationRegime();

    if el == EL3 then
        tcf = SCTLR_EL3.TCF;
    elsif el == EL2 then
        if AArch64.AccessUsesEL(acctype) == EL0 then
            tcf = SCTLR_EL2.TCF0;
        else
            tcf = SCTLR_EL2.TCF;
    elsif el == EL1 then
        if AArch64.AccessUsesEL(acctype) == EL0 then
            tcf = SCTLR_EL1.TCF0;
        else
            tcf = SCTLR_EL1.TCF;

    if tcf == '11' then //reserved value
        if !HaveMTE3Ext() then
            (-, tcf) = ConstrainUnpredictableBits(Unpredictable_RESTCF);

    return tcf;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// =====
AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
// External aborts on instruction fetch must be taken synchronously
if

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
// External aborts on instruction fetch must be taken synchronously
if HaveDoubleFaultExt() then assert fault.statuscode != Fault_AsyncExternal;
route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
route_to_el2 = ( route_to_el2 = (PSTATE.EL IN {EL2Enabled()} && PSTATE.EL IN {EL0, EL1} &&
(HCR_EL2.TGE == '1' ||
) && EL2Enabled() &&
(HCR_EL2.TGE == '1' ||
(HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) || (fault)) ||
IsSecondStage(fault)));

bits(64) preferred_exception_return =
IsSecondStage(fault));

bits(64) preferred_exception_return = ThisInstrAddr();

if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
    vect_offset = 0x180;
else
    vect_offset = 0x0;

exception = exception = AArch64.AbortSyndrome( AArch64.AbortSyndrome(Exception_InstructionAbort,
bits(2) target_el = if PSTATE.EL == EL1;
if PSTATE.EL == EL3 || route_to_el3 then
    target_el = || route_to_el3 then AArch64.TakeException(EL3;
, exception, preferred_exception_return, vect_offset);
elseif PSTATE.EL == EL2 || route_to_el2 then
    target_el = || route_to_el2 then AArch64.TakeException(EL2; , exception, preferred_exception_return
else
    AArch64.TakeException(EL1(target_el, exception, preferred_exception_return, vect_offset); , exception,
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// =====
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_PCAlignment);
exception.vaddress = ThisInstrAddr();

bits(2) target_el = if EL1;
if UInt(PSTATE.EL) > UInt(EL1) then
    target_el = PSTATE.EL;
elseif) then AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elseif EL2Enabled() && HCR_EL2.TGE == '1' then
    target_el =() && HCR_EL2.TGE == '1' then AArch64.TakeException(EL2; , exception, preferred_excepti
else
    AArch64.TakeException(EL1(target_el, exception, preferred_exception_return, vect_offset); , exception,
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.RaiseTagCheckFault

```
// AArch64.RaiseTagCheckFault()
// =====
// Raise a tag check fault exception.

AArch64.RaiseTagCheckFault(bits(64) va, boolean write)
    bits(2) target_el;
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset = 0x0;

    exception = if PSTATE.EL == ExceptionSyndromeEL0 then
        target_el = if HCR_EL2.TGE == '0' then Exception_DataAbort;
    exception.syndrome<5:0> = '010001';
    if write then
        exception.syndrome<6> = '1';
    exception.vaddress = bits(4) UNKNOWN : va<59:0>;

    bits(2) target_el = EL1;
    ifelse UIntEL2(PSTATE.EL) >;
    else
        target_el = PSTATE.EL;

    exception = UIntExceptionSyndrome(EL1Exception_DataAbort) then
        target_el = PSTATE.EL;
    elseif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;};
    exception.syndrome<5:0> = '010001';
    if write then
        exception.syndrome<6> = '1';
    exception.vaddress = bits(4) UNKNOWN : va<59:0>;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.ReportTagCheckFault

```
// AArch64.ReportTagCheckFault()
// =====
// Records a tag check fault exception into the appropriate TCFR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    if el == EL3 then
        assert ttbr == '0';
        TFSR_EL3.TF0 = '1';
    elseif el == EL2 then
        if ttbr == '0' then
            TFSR_EL2.TF0 = '1';
        else
            TFSR_EL2.TF1 = '1';
    elseif el == EL1 then
        if ttbr == '0' then
            TFSR_EL1.TF0 = '1';
        else
            TFSR_EL1.TF1 = '1';
    elseif el == EL0 then
        if ttbr == '0' then
            TFSRE0_EL1.TF0 = '1';
        else
            TFSRE0_EL1.TF1 = '1';
```


Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// =====
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SPAlignment);

    bits(2) target_el = if EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif then AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = () && HCR_EL2.TGE == '1' then AArch64.TakeException(EL2; , exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1(target_el, exception, preferred_exception_return, vect_offset); , exception,
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.TagCheckFault

```
// AArch64.TagCheckFault()
// =====
// Handle a tag check fault condition.

AArch64.TagCheckFault(bits(64) vaddress, AccType acctype, boolean iswrite)
    bits(2) tcf, el;
    el = bits(2) tcf = AArch64.AccessUsesEL(acctype);
    tcf = AArch64.EffectiveTCF(acctype);
    case tcf of
        when '00' // Tag Check Faults have no effect on the PE
            return;
        when '01' // Tag Check Faults cause a synchronous exception
            AArch64.RaiseTagCheckFault(vaddress, iswrite);
        when '10' // Tag Check Faults are asynchronously accumulated
            AArch64.ReportTagCheckFault(el, vaddress<55>);
(PSTATE.EL, vaddress<55>);
        when '11' // Tag Check Faults cause a synchronous exception on reads or on
            // a read-write access, and are asynchronously accumulated on writes
            // Check for access performing both a read and a write.
            readwrite = acctype IN {AccType_ATOMICRW,
                                   AccType_ORDEREDATOMICRW,
                                   AccType_ORDEREDRW};

            if !iswrite || readwrite then
                AArch64.RaiseTagCheckFault(vaddress, iswrite);
            else
                AArch64.ReportTagCheckFault(PSTATE.EL, vaddress<55>);
```

Library pseudocode for aarch64/exceptions/aborts/BranchTargetException

```
// BranchTargetException()
// =====
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
    bits(64) preferred_exception_return =
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_BranchTarget);
    exception.syndrome<1:0> = PSTATE.BTYPE;
    exception.syndrome<24:2> = Zeros(); // RES0

    bits(2) target_el = if EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 then AArch64.TakeException && (PSTATE.EL, exception, preferred_exception_return
    elsif route_to_el2 then EL2EnabledAArch64.TakeException() && HCR_EL2.TGE == '1' then
        target_el = EL2; exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1(target_el, exception, preferred_exception_return, vect_offset); exception,
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// =====

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
    exception = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====

AArch64.TakePhysicalSErrorException(bits(25) syndrome)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    bits(2) target_el;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

    if IsSErrorEdgeTriggered(target_el, syndrome) then
        ClearPendingPhysicalSError();

    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome = syndrome;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =====

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1'; // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;

    exception = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =====

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1'; // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// =====

AArch64.TakeVirtualSErrorException(bits(25) syndrome)

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1'; // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;
    exception = ExceptionSyndrome(Exception_SError);

    if HaveRASExt() then
        exception.syndrome<24> = VESR_EL2.IDS;
        exception.syndrome<23:0> = VESR_EL2.ISS;
    else
        impdef_syndrome = syndrome<24> == '1';
        if impdef_syndrome then exception.syndrome = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =====

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// =====

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                    EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// =====

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
    exception.syndrome<5:0> = '100010'; // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// =====
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =====

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x0;

    if HaveNV2Ext\(\) && fault.acctype == AccType\_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception\_NV2Watchpoint, fault, vaddress);
    else
        exception = AArch64.AbortSyndrome(Exception\_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il_is_valid = TRUE;
    from_32 = UsingAArch32();

    case exceptype of
        when Exception_Uncategorized          ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap                ec = 0x01;
        when Exception_CP15RRTTrap            ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap            ec = 0x04; assert from_32;
        when Exception_CP14RRTTrap            ec = 0x05; assert from_32;
        when Exception_CP14DTTrap             ec = 0x06; assert from_32;
        when Exception_AdvSIMDFPAccessTrap    ec = 0x07;
        when Exception_FPIDTrap               ec = 0x08;
        when Exception_PACTrap                 ec = 0x09;
        when Exception_LDST64BTrap            ec = 0x0A;
        when Exception_CP14RRTTrap            ec = 0x0C; assert from_32;
        when Exception_BranchTarget           ec = 0x0D;
        when Exception_IllegalState           ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall         ec = 0x11;
        when Exception_HypervisorCall         ec = 0x12;
        when Exception_MonitorCall            ec = 0x13;
        when Exception_SystemRegisterTrap     ec = 0x18; assert !from_32;
        when Exception_SVEAccessTrap          ec = 0x19; assert !from_32;
        when Exception_ERetTrap               ec = 0x1A; assert !from_32;
        when Exception_PACFail                ec = 0x1C; assert !from_32;
        when Exception_InstructionAbort       ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAlignment            ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort              ec = 0x24;
        when Exception_NV2DataAbort           ec = 0x25;
        when Exception_SPAlignment            ec = 0x26; il_is_valid = FALSE; assert !from_32;
        when Exception_FPtrappedException     ec = 0x28;
        when Exception_SError                 ec = 0x2F; il_is_valid = FALSE;
        when Exception_Breakpoint             ec = 0x30; il_is_valid = FALSE;
        when Exception_SoftwareStep           ec = 0x32; il_is_valid = FALSE;
        when Exception_Watchpoint             ec = 0x34; il_is_valid = FALSE;
        when Exception_NV2Watchpoint          ec = 0x35; il_is_valid = FALSE;
        when Exception_SoftwareBreakpoint     ec = 0x38;
        when Exception_VectorCatch            ec = 0x3A; il_is_valid = FALSE; assert from_32;
        otherwise                             Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    assert from_32 || il == '1'; // AArch64 instructions always 32-bit

    return (ec,il);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =====
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = exception.syndrome;
    iss2 = exception.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = (Zeros(27) :    // <63:37>
                     iss2          :    // <36:32>
                     ec<5:0>       :    // <31:26>
                     il            :    // <25>
                     iss);         // <24:0>

    if exceptype IN {
        Exception_InstructionAbort,
        Exception_PCAlignment,
        Exception_DataAbort,
        Exception_NV2DataAbort,
        Exception_NV2Watchpoint,
        Exception_Watchpoint
    } then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if exception.ipavalid then
        HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
        if IsSecureEL2Enabled() && IsSecure() then
            HPFAR_EL2.NS = exception.NS;
        else
            HPFAR_EL2.NS = '0';
    elsif target_el == EL2 then
        HPFAR_EL2<43:4> = bits(40) UNKNOWN;

    return;
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch64.ResetControlRegisters(boolean cold_reset);
```


Library pseudocode for aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// =====
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
assert assert ! HaveAArch64HighestELUsingAArch32();

// Enter the highest implemented Exception level in AArch64 state
PSTATE.nRW = '0';
if HaveEL(EL3) then
    PSTATE.EL = EL3;
elseif HaveEL(EL2) then
    PSTATE.EL = EL2;
else
    PSTATE.EL = EL1;

// Reset System registers and other system components // Reset the system registers and other syst
AArch64.ResetControlRegisters(cold_reset);

// Reset all other PSTATE fields
PSTATE.SP = '1'; // Select stack pointer
PSTATE.<D,A,I,F> = '1111'; // All asynchronous exceptions masked
PSTATE.SS = '0'; // Clear software step bit
PSTATE.DIT = '0'; // PSTATE.DIT is reset to 0 when resetting into AArch64
PSTATE.IL = '0'; // Clear Illegal Execution state bit

// All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
// below are UNKNOWN bitstrings after reset. In particular, the return information registers
// ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
// is impossible to return from a reset in an architecturally defined way.
AArch64.ResetGeneralRegisters();
AArch64.ResetSIMDFPRegisters();
AArch64.ResetSpecialRegisters();
ResetExternalDebugRegisters(cold_reset);

bits(64) rv; // IMPLEMENTATION DEFINED reset vector

if HaveEL(EL3) then
    rv = RVBAR_EL3;
elseif HaveEL(EL2) then
    rv = RVBAR_EL2;
else
    rv = RVBAR_EL1;

// The reset vector must be correctly aligned
assert IsZero(rv<63:AArch64.PAMax()>) && IsZero(rv<1:0>);

boolean branch_conditional = FALSE;
BranchTo(rv, BranchType_RESET, branch_conditional);
```

Library pseudocode for aarch64/exceptions/ieeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// =====

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome\(Exception\_FPTrappedException\);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            exception.syndrome<23> = '1'; // TFV
        else
            exception.syndrome<23> = '0'; // TFV
    else
        exception.syndrome<23> = '1'; // TFV
    exception.syndrome<10:8> = bits(3) UNKNOWN; // VECITR
    if exception.syndrome<23> == '1' then
        exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOf
    else
        exception.syndrome<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled\(\) && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x0;

    if UInt\(PSTATE.EL\) > UInt\(EL1\) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// =====
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL\(EL2\);

    if UsingAArch32\(\) then AArch32.ITAdvance\(\);
    SSAdvance\(\);
    bits(64) preferred_exception_return = NextInstrAddr\(\);
    vect_offset = 0x0;

    exception = ExceptionSyndrome\(Exception\_HypervisorCall\);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// =====

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_MonitorCall);
    exception.syndrome<15:0> = immediate;

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// =====
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```



```

// AArch64.TakeException()
// =====
// Take an exception to an Exception level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception,
                      bits(64) preferred_exception_return, integer vect_offset)
assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

if HaveIESB() then
    sync_errors = SCTLR[target_el].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR\_EL3.<EA,NMEA> == '11' && target_el == EL3);
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();
        iesb_req = FALSE;
        sync_errors = FALSE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
else
    sync_errors = FALSE;

SynchronizeContext();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
MaybeZeroSVEUppers(target_el);

if UInt(target_el) > UInt(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAArch32(EL2);
        else
            lower_32 = ELUsingAArch32(EL1);
    elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
        lower_32 = ELUsingAArch32(EL0);
    else
        lower_32 = ELUsingAArch32(target_el - 1);
    vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

elsif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;

bits(64) spsr = GetPSRFromPSTATE(AArch64\_NonDebugState);

if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
    if HaveNV2Ext() && (HCR\_EL2.<NV,NV1,NV2> == '100' || HCR\_EL2.<NV,NV1,NV2> == '111') then
        spsr<3:2> = '10';
    else
        if HaveNVExt() && HCR\_EL2.<NV,NV1> == '10' then
            spsr<3:2> = '10';

if HaveBTIExt() && !UsingAArch32() then
    // SPSR[].BTYP is only guaranteed valid for these exception types
    if exception.exceptype IN {Exception\_SError, Exception\_IRQ, Exception\_FIQ,
                             Exception\_SoftwareStep, Exception\_PCAlignment,
                             Exception\_InstructionAbort, Exception\_Breakpoint,
                             Exception\_VectorCatch, Exception\_SoftwareBreakpoint,
                             Exception\_IllegalState, Exception\_BranchTarget} then
        zero_btype = FALSE;
    else
        zero_btype = ConstrainUnpredictableBool(Unpredictable\_ZEROBTYP);
    if zero_btype then spsr<11:10> = '00';

if HaveNV2Ext() && exception.exceptype == Exception\_NV2DataAbort && target_el == EL3 then
    // External aborts are configured to be taken to EL3
    exception.exceptype = Exception\_DataAbort;
if !(exception.exceptype IN {Exception\_IRQ, Exception\_FIQ}) then
    AArch64.ReportException(exception, target_el);

```

```

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR[] = spsr;
ELR[] = preferred_exception_return;

PSTATE.SS = '0';
PSTATE.<D,A,I,F> = '1111';
PSTATE.IL = '0';
if from_32 then // Coming from AArch32
    PSTATE.IT = '00000000';
    PSTATE.T = '0'; // PSTATE.J is RES0
if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
    SCTRL[].SPAN == '0') then
    PSTATE.PAN = '1';
if HaveUA0Ext() then PSTATE.UA0 = '0';
if HaveBTIExt() then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = SCTRL[].DSSBS;
if HaveMTEExt() then PSTATE.TCO = '1';

boolean branch_conditional = FALSE;
BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION, branch_conditional);, branch_conditional

if sync_errors then

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    EndOfInstruction();

```

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

```

// AArch64.AArch32SystemAccessTrap()
// =====
// Trapped AARCH32 system register access.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

```



```

// AArch64.AArch32SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;

    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTTrap);
        when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTTrap);
        when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros();

    if exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTTrap, Exception_CP15RTTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if exception.exceptype != Exception_FPIDTrap then // When trap is not for VMRS
            iss<19:17> = instr<7:5>; // opc2
            iss<16:14> = instr<23:21>; // opc1
            iss<13:10> = instr<19:16>; // CRn
            iss<4:1> = instr<3:0>; // CRm
        else
            iss<19:17> = '000';
            iss<16:14> = '111';
            iss<13:10> = instr<19:16>; // reg
            iss<4:1> = '0000';

            if instr<20> == '1' && instr<15:12> == '1111' then // MRC, Rt==15
                iss<9:5> = '11111';
            elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
                iss<9:5> = bits(5) UNKNOWN;
            else
                iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        elsif exception.exceptype IN {Exception_CP14RRTTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTTrap} then
            // Trapped MRRC/MCRR, VMRS/VMSR
            iss<19:16> = instr<7:4>; // opc1
            if instr<19:16> == '1111' then // Rt2==15
                iss<14:10> = bits(5) UNKNOWN;
            else
                iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

            if instr<15:12> == '1111' then // Rt==15
                iss<9:5> = bits(5) UNKNOWN;
            else
                iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
            iss<4:1> = instr<3:0>; // CRm
        elsif exception.exceptype == Exception_CP14DTTTrap then
            // Trapped LDC/STC
            iss<19:12> = instr<7:0>; // imm8
            iss<4> = instr<23>; // U
            iss<2:1> = instr<24,21>; // P,W
            if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
                iss<9:5> = bits(5) UNKNOWN;
                iss<3> = '1';
            elsif exception.exceptype == Exception_Uncategorized then
                // Trapped for unknown reason
                iss<9:5> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rn
                iss<3> = '0';

            iss<0> = instr<20>; // Direction

            exception.syndrome<24:20> = ConditionSyndrome();
            exception.syndrome<19:0> = iss;

```



```
return exception;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// =====
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        exception = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained AArch32 traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2 and HCR_EL2.
// Check for coarse-grained AArch32 CP15 traps in HSTR_EL2 and HCR_EL2.

boolean AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR_EL2<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR_EL2.TIDCP
        if (HCR_EL2.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
             (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// =====
// Check against CPACR[]

AArch64.CheckFPAdvSIMDEnabled() AArch64.CheckFPAdvSIMDEnabled()
    if PSTATE.EL IN {
        , EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap AArch64.CheckFPEnabledEL0();(); // Also check against CPTR_EL
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

    return;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPEEnabled

```
// AArch64.CheckFPEEnabled()
// =====
// Check against CPACR[]

AArch64.CheckFPEEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap(); // Also check against CPTR_EL2 and CPTR_EL3
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

```
// AArch64.CheckForERetTrap()
// =====
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETAA, ERETAB when either HCR_EL2.NV or HFGITR_EL2.ERET is set,
    // is trapped to EL2
    route_to_el2 = (PSTATE.EL == EL1 && EL2Enabled() &&
        ((HaveNVExt() && HCR_EL2.NV == '1') ||
        (HaveFGTExt() && HCR_EL2.<E2H, TGE> != '11' &&
        (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ERET == '1')));
    if route_to_el2 then
        ExceptionRecord exception;
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_ERetTrap);
        if !eret_with_pac then // ERET
            exception.syndrome<1> = '0';
            exception.syndrome<0> = '0'; // RES0
        else
            exception.syndrome<1> = '1';
            if pac_uses_key_a then // ERETAA
                exception.syndrome<0> = '0';
            else // ERETAB
                exception.syndrome<0> = '1';
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSMCUnDefOrTrap

```
// AArch64.CheckForSMCUnDefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUnDefOrTrap(bits(16) imm)
    if PSTATE.EL == EL0 then UNDEFINED;
    if (!(PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
        HaveEL(EL3) && SCR_EL3.SMD == '1') then
        UNDEFINED;
    route_to_el2 = FALSE;
    if !HaveEL(EL3) then
        if PSTATE.EL == EL1 && EL2Enabled() then
            if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UNDEFINED;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSVCTrap

```
// AArch64.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL0) && !ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' && (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        elsif PSTATE.EL == EL1 then
            route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' && (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        if route_to_el2 then
            exception = ExceptionSyndrome(Exception_SupervisorCall);
            exception.syndrome<15:0> = immediate;
            bits(64) preferred_exception_return = ThisInstrAddr();
            vect_offset = 0x0;

            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFXTrap

```
// AArch64.CheckForWFXTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFXTrap(bits(2) target_el, WFXType wfxtype)
    assert HaveEL(target_el);

    boolean is_wfe = wfxtype IN {WFXType_WFE, WFXType_WFET};
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';

    if trap then
        AArch64.WFXTrap(wfxtype, target_el);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;

        exception = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =====
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrap()
// =====
// Trapped access to AArch64 system register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

```
// AArch64.SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;
    case ec of
        when 0x0 // Trapped access due to unknown reason
            exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7 // Trapped access to SVE, Advance SIMD
            exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
        when 0x18 // Trapped access to system register
            exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
            instr = ThisInstr();
            exception.syndrome<21:20> = instr<20:19>; // Op0
            exception.syndrome<19:17> = instr<7:5>; // Op2
            exception.syndrome<16:14> = instr<18:16>; // Op1
            exception.syndrome<13:10> = instr<15:12>; // CRn
            exception.syndrome<9:5> = instr<4:0>; // Rt
            exception.syndrome<4:1> = instr<11:8>; // CRm
            exception.syndrome<0> = instr<21>; // Direction
        when 0x19 // Trapped access to SVE System register
            exception = ExceptionSyndrome(Exception_SVEAccessTrap);
        otherwise
            Unreachable();

    return exception;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.UndefinedFault

```
// AArch64.UndefinedFault()
// =====

AArch64.UndefinedFault()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =====

AArch64.WFxTrap(WFxType wfxtype, bits(2) target_el)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();

    case wfxtype of
        when WFxType_WFI
            exception.syndrome<1:0> = '00';
        when WFxType_WFE
            exception.syndrome<1:0> = '01';
        when WFxType_WFIT
            exception.syndrome<1:0> = '10';
            if when HaveFeatWFXT2() then
                exception.syndrome<2> = '1'; // Register field is valid
                exception.syndrome<9:5> = ThisInstr().<4:0>;
            else
                exception.syndrome<2> = '0'; // Register field is invalid
        when WFxType_WFET
            exception.syndrome<1:0> = '11';
            if HaveFeatWFXT2() then
                exception.syndrome<2> = '1'; // Register field is valid
                exception.syndrome<9:5> = ThisInstr().<4:0>;
            else
                exception.syndrome<2> = '0'; // Register field is invalid
    exception.syndrome<1:0> = '11';

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =====
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

Library pseudocode for aarch64/exceptions/traps/CheckFPEnabled64CheckLDST64BEnabled

```
// CheckFPEnabled64()
// =====
// AArch64 instruction wrapper// CheckLDST64BEnabled()
// =====
// Checks for trap on ST64B and LD64B instructions

CheckFPEnabled64()CheckLDST64BEnabled()
    boolean trap = FALSE;
    bits(25) iss =
    ('10'); // 0x2

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTL_EL1.EnALS == '0';
            target_el = if HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTL_EL2.EnALS == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() && HaveFeatHGX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnALS == '0';
        target_el = EL2;

    if trap then LDST64BTrapAArch64.CheckFPEnabledZeroExtend();(target_el, iss);
```

Library pseudocode for aarch64/exceptions/traps/CheckLDST64BEnabledCheckST64BV0Enabled

```
// CheckLDST64BEnabled()
// CheckST64BV0Enabled()
// =====
// Checks for trap on ST64B and LD64B instructions// Checks for trap on ST64BV0 instruction

CheckLDST64BEnabled()
CheckST64BV0Enabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('10'); // 0x2
    ('1'); // 0x1

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTL_EL1.EnALS == '0';
            target_el = if trap = SCTL_EL1.EnAS0 == '0';
            target_el = if HCR_EL2.TGE == '1' then EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTL_EL2.EnALS == '0';
            trap = SCTL_EL2.EnAS0 == '0';
            target_el = EL2;
    else
        target_el =
        if (!trap && EL1;

    if (!trap && EL2Enabled() && HaveFeatHGX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnALS == '0';
        () || HCRX_EL2.EnAS0 == '0';
        target_el = EL2;

    if !trap && PSTATE.EL != EL3 then
        trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
        target_el = EL3;

    if trap then LDST64BTrap(target_el, iss);
```

Library pseudocode for aarch64/exceptions/traps/CheckST64BV0EnabledCheckST64BVEnabled

```
// CheckST64BV0Enabled()
// =====
// Checks for trap on ST64BV0 instruction// CheckST64BVEnabled()
// =====
// Checks for trap on ST64BV instruction

CheckST64BV0Enabled()
CheckST64BVEnabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtendZeros('1'); // 0x1
();

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnAS0 == '0';
            target_el = if trap == SCTLR_EL1.EnASR == '0';
            target_el = if HCR_EL2.TGE == '1' then EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnAS0 == '0';
            trap = SCTLR_EL2.EnASR == '0';
            target_el = EL2;

        if (!trap && EL2Enabled() && HaveFeatHCX() &&
            ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
            trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnAS0 == '0';
            HCRX_EL2.EnASR == '0';
            target_el = EL2;

        if !trap && PSTATE.EL != EL3 then
            trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
            target_el = EL3;

    if trap then LDST64BTrap(target_el, iss);
```

Library pseudocode for aarch64/exceptions/traps/CheckST64BVEnabled

```
// CheckST64BVEnabled()
// =====
// Checks for trap on ST64BV instruction

CheckST64BVEnabled()
    boolean trap = FALSE;
    bits(25) iss = Zeros();

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnASR == '0';
            target_el = EL2;

        if (!trap && EL2Enabled() && HaveFeatHCX() &&
            ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
            trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnASR == '0';
            target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```


Library pseudocode for aarch64/exceptions/traps/LDST64BTrap

```
// LDST64BTrap()
// =====
// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions

LDST64BTrap(bits(2) target_el, bits(25) iss)
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_LDST64BTrap);
    exception.syndrome = iss;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
```

Library pseudocode for aarch64/exceptions/traps/WFETrapDelay

```
// WFETrapDelay()
// =====
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

(boolean, integer) WFETrapDelay(bits(2) target_el)
    case target_el of
        when EL1
            if !IsInHost() then
                delay_enabled = SCTLR_EL1.TWEDEn == '1';
                delay = 1 << (UInt(SCTLR_EL1.TWEDEL) + 8);
            else
                delay_enabled = SCTLR_EL2.TWEDEn == '1';
                delay = 1 << (UInt(SCTLR_EL2.TWEDEL) + 8);
        when EL2
            assert(delay_enabled = HCR_EL2.TWEDEn == '1');
            delay = 1 << (EL2Enabled());
            delay_enabled = HCR_EL2.TWEDEn == '1';
            delay = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
        when EL3
            delay_enabled = SCR_EL3.TWEDEn == '1';
            delay = 1 << (UInt(SCR_EL3.TWEDEL) + 8);

    return (delay_enabled, delay);
```

Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelay

```
// WaitForEventUntilDelay()
// =====
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.

boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay);
    boolean eventarrived = FALSE;
    // set eventarrived to TRUE if WaitForEvent() returns before
    // 'delay' expires when delay_enabled is TRUE.
    return eventarrived;
```

Library pseudocode for aarch64/functions/aborts/AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// =====
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception level using AArch64.

(bits(25), bits(5)) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    bits(25) iss = Zeros\(\);
    bits(5) iss2 = Zeros\(\);

    if !HaveFeatLS64\(\) && HaveRASExt\(\) && IsAsyncAbort(fault) then
        iss<12:11> = fault.errortype; // SET

    if d_side then
        if HaveFeatLS64\(\) && fault.acctype == AccType\_ATOMICLS64 then
            if (fault.statuscode IN {Fault\_AccessFlag,
                Fault\_Translation, Fault\_Permission}) then
                (iss2, iss<24:14>, iss<12:11>) = LS64InstructionSyndrome\(\);
            else
                if (IsSecondStage(fault) && !fault.s2fslwalk &&
                    (!IsExternalSyncAbort(fault) ||
                     (!HaveRASExt\(\) && fault.acctype == AccType\_TTW &&
                      boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")))) then
                    iss<24:14> = LSInstructionSyndrome\(\);

        if HaveNV2Ext\(\) && fault.acctype == AccType\_NV2REGISTER then
            iss<13> = '1'; // Fault is generated by use of VNCR_EL2

        if fault.acctype IN {AccType\_DC, AccType\_IC, AccType\_AT, AccType\_ATPAN} then
            iss<8> = '1'; iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';

        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fslwalk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return (iss, iss2);
```

Library pseudocode for aarch64/functions/aborts/LS64InstructionSyndrome

```
// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field, and LST.
(bits(5), bits(11), bits(2)) LS64InstructionSyndrome\(\);
```

Library pseudocode for aarch64/functions/cache/AArch64.DataMemZero

```
// AArch64.DataMemZero()
// =====
// Write Zero to data memory

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AddressDescriptor memaddrdesc, integer size)
    iswrite = TRUE;
    for i = 0 to size-1
        accdesc = CreateAccessDescriptor(AccType_DCZVA);
        if HaveMTEExt() then
            if AArch64.AccessIsTagChecked(vaddress, AccType_DCZVA) then
                bits(4) ptag = AArch64.PhysicalTag(vaddress);
                if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                    if boolean IMPLEMENTATION_DEFINED "DC_ZVA tag fault reported with lowest faulting address" then
                        AArch64.TagCheckFault(vaddress, AccType_DCZVA, iswrite);
                    else
                        AArch64.TagCheckFault(regval, AccType_DCZVA, iswrite);
                memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros());
                if IsFault(memstatus) then
                    HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return;
```

Library pseudocode for aarch64/functions/cache/AArch64.TagMemZero

```
// AArch64.TagMemZero()
// =====
// Write Zero to tag memory

AArch64.TagMemZero(bits(64) vaddress, integer size)
    integer count = size >> LOG2_TAG_GRANULE;
    bits(4) tag = AArch64.AllocationTagFromAddress(vaddress);
    for i = 0 to count-1
        AArch64.MemTag[vaddress, AccType_NORMAL] = tag;
        vaddress = vaddress + TAG_GRANULE;
    return;
```

Library pseudocode for aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareability != Shareability_NSH then
            passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    return passed;
```

Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

Library pseudocode for aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)
    acctype = AccType\_ATOMIC;
    iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// =====

bits(N) FPRSqrtStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    FPCRTType fpcr = FPCR[];
    op1 = FPNeg(op1);
    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00'; // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, FALSE, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPType\_Infinity);
        inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);
        zero2 = (type2 == FPType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0');
        elsif inf1 || inf2 then
            result = FPIfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);

    return result;
```

Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// =====

bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
  assert N IN {16, 32, 64};
  bits(N) result;
  FPCRTType fpcr = FPCR[];
  op1 = FPNeg(op1);

  boolean altfp = HaveAltFP() && fpcr.AH == '1';
  boolean fpexc = !altfp; // Generate no floating-point exceptions
  if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
  if altfp then fpcr.RMode = '00'; // Use RNE rounding mode

  (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
  (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, FALSE, fpexc);
  FPRounding rounding = FPRoundingMode(fpcr);

  if !done then
    inf1 = (type1 == FPType\_Infinity);
    inf2 = (type2 == FPType\_Infinity);
    zero1 = (type1 == FPType\_Zero);
    zero2 = (type2 == FPType\_Zero);

    if (inf1 && zero2) || (zero1 && inf2) then
      result = FPTwo('0');
    elsif inf1 || inf2 then
      result = FPInfinity(sign1 EOR sign2);
    else
      // Fully fused multiply-add
      result_value = 2.0 + (value1 * value2);
      if result_value == 0.0 then
        // Sign of exact zero result depends on rounding mode
        sign = if rounding == FPRounding\_NEGINF then '1' else '0';
        result = FPZero(sign);
      else
        result = FPRound(result_value, fpcr, rounding, fpexc);

  return result;
```

Library pseudocode for aarch64/functions/memory/AArch64.AccessIsTagChecked

```
// AArch64.AccessIsTagChecked()
// =====
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccType acctype)
    if PSTATE.M<4> == '1' then return FALSE;

    if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
        return FALSE;

    if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') then
        return FALSE;

    if !AArch64.AllocationTagAccessIsEnabled(acctype) then
        return FALSE;

    if acctype IN {AccType\_IFETCH, AccType\_TTW, AccType\_DC, AccType\_IC} then
        return FALSE;

    if acctype == AccType\_NV2REGISTER then
        return FALSE;

    if PSTATE.TC0 == '1' then
        return FALSE;

    if !IsTagCheckedInstruction() then
        return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/memory/AArch64.AddressWithAllocationTag

```
// AArch64.AddressWithAllocationTag()
// =====
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, AccType acctype, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag;
    if AArch64.AllocationTagAccessIsEnabled(acctype) then
        tag = allocation_tag;
    else
        tag = '0000';
    result<59:56> = tag;
    return result;
```

Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagFromAddress

```
// AArch64.AllocationTagFromAddress()
// =====
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
    return tagged_address<59:56>;
```

Library pseudocode for aarch64/functions/memory/AArch64.CheckAlignment

```
// AArch64.CheckAlignment()
// =====

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype,
                               boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
                          AccType_ORDEREDATOMICRW, AccType_ATOMICS64, AccType_A32LSMD };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC };
    vector = acctype == AccType_VEC;
    if SCTLRL[0].A == '1' then check = TRUE;
    elsif HavelSE2Ext() then
        check = (UInt(address<0+:4>) + alignment > 16) && ((ordered && SCTLRL[0].nAA == '0') || atomic);
    else check = atomic || ordered;

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

Library pseudocode for aarch64/functions/memory/AArch64.CheckTag

```
// AArch64.CheckTag()
// =====
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean AArch64.CheckTag(AddressDescriptor memaddrdesc, bits(4) ptag, boolean write)
    if memaddrdesc.memattrs.tagged then
        return ptag == MemTag[memaddrdesc];
    else
        return TRUE; AArch64.CheckTag(AddressDescriptor memaddrdesc, AccessDescriptor accdesc, bits(4) ptag)
    if memaddrdesc.memattrs.tagged then
        (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        return ptag == readtag;
    else
        return TRUE;
```



```

// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned]
acctype, boolean wasaligned]
    boolean ispair = FALSE;
    return AArch64.MemSingle[address, size, acctype, aligned, ispair];
[address, size, acctype, wasaligned, ispair];

// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean
acctype, boolean wasaligned, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    if HaveLSE2Ext() then
        assert CheckAllInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, size);(address, size);

AddressDescriptor memaddrdesc;
bits(size*8) value;
iswrite = FALSE;

memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
accdesc =

AddressDescriptor memaddrdesc;
bits(size*8) value;
iswrite = FALSE;

memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTE2Ext() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if ! if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then AArch64.CheckTag(memaddr
        AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

integer halfsize = size DIV 2;
(atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswr
(address, memaddrdesc.memattrs, size, acctype, iswrite, wasaligned, ispair);
if atomic then
    (memstatus, value) = value = _Mem[memaddrdesc, size, accdesc, FALSE];
elseif splitpair then
    assert ispair;
    value1 = _Mem[memaddrdesc, halfsize, accdesc, FALSE];
    memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize<52-1:0>;
    value2 = _Mem[memaddrdesc, halfsize, accdesc, FALSE];

    value = value2<8*(size DIV 2)-1:0>;value1<8*(size DIV 2)-1:0>;
else
    for i = 0 to size-1
        value<8*i+7:8*i> = _Mem[memaddrdesc, 1, accdesc, FALSE];
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1<52-1:0>;

return value;

```

```

// AArch64.MemSingle[] - assignment (write) form
// ===== PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    elsif splitpair then
        assert ispair;
        (memstatus, value1) = PhysMemRead(memaddrdesc, halfsize, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize<52-1:0>;
        (memstatus, value2) = PhysMemRead(memaddrdesc, halfsize, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);

        value = value2<8*(size DIV 2)-1:0>;value1<8*(size DIV 2)-1:0>;
    else
        for i = 0 to size-1
            (memstatus, value<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1<52-1:0>;
        return value;

// AArch64.MemSingle[] - assignment (write) form
// =====

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value
acctype, boolean wasaligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
[address, size, acctype, wasaligned, ispair] = value;
    return;

// AArch64.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bits
acctype, boolean wasaligned, boolean ispair] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    if HaveLSE2Ext() then
        assert CheckAllInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, size);{address, size};

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability !=

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH thenthen
        ClearExclusiveByAddress(memaddrdesc.paddress,
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access

```

```

accdesc = CreateAccessDescriptor(acctype);
if HaveMTE2Ext() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then AArch64.CheckTag(memaddr
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

    (atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswr
    if atomic then
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    elsif splitpair then
        assert ispair;
        integer halfsize = size DIV 2;
        bits(halfsize*8) val1 = value<(8*halfsize)-1:0>;
        bits(halfsize*8) val2 = value<(16*halfsize)-1:(8*halfsize)>;
        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, val1);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize<52-1:0>;
        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, val2);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
    else
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
(address, memaddrdesc.memattrs, size, acctype, iswrite, wasaligned, ispair);
    if atomic then
        Mem[memaddrdesc, size, accdesc] = value;
    elsif splitpair then
        assert ispair;
        integer halfsize = size DIV 2;
        bits(halfsize*8) val1 = value<(8*halfsize)-1:0>;
        bits(halfsize*8) val2 = value<(16*halfsize)-1:(8*halfsize)>;
        Mem[memaddrdesc, halfsize, accdesc] = val1;
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize<52-1:0>;
        Mem[memaddrdesc, halfsize, accdesc] = val2;
    else
        for i = 0 to size-1
            Mem[memaddrdesc, 1, accdesc] = value<8*i+7:8*i>;
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1<52-1:0>;
return;

```



```

// AArch64.MemTag[] - non-assignment (read) form
// =====
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccType acctype]
    assert acctype == AccType_NORMAL;;
    AddressDescriptor memaddrdesc;
    bits(4) value;

    iswrite = FALSE;
    aligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned,
    AddressDescriptor memaddrdesc;
    bits(4) value;

    iswrite = FALSE;
    aligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned,
    TAG_GRANULE);
    accdesc = // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Return the granule tag if tagging is enabled...
    if CreateAccessDescriptor(acctype);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Return the granule tag if tagging is enabled...
    if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
        (memstatus, tag) = return _MemTag[memaddrdesc];
    else
        // ...otherwise read tag as zero.
        return '0000';

// AArch64.MemTag[] - assignment (write) form
// =====
// Store an Allocation Tag to memory. PhysMemTagRead(memaddrdesc, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
    return tag;
    else
        // ...otherwise read tag as zero.
        return '0000';

// AArch64.MemTag[] - assignment (write) form
// =====
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccType acctype] = bits(4) value
    assert acctype == AccType_NORMAL;;
    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // Stores of allocation tags must be aligned
    if address !=
        AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // Stores of allocation tags must be aligned
    if address != Align(address, TAG_GRANULE) then
        boolean secondstage = FALSE; boolean secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    aligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned,
    AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    aligned = TRUE;

```

```

memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned,
                                         TAG_GRANULE);
accdesc =
// It is CONSTRAINED UNPREDICTABLE if tags stored to memory locations marked as Device
// generate an Alignment Fault or store the data to locations.
if memaddrdesc.memattrs.memtype == CreateAccessDescriptor(acctype);

// It is CONSTRAINED UNPREDICTABLE if tags stored to memory locations marked as Device
// generate an Alignment Fault or store the data to locations.
if memaddrdesc.memattrs.memtype == MemType_Device then
    c = ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE);
    assert c IN {Constraint_NONE, Constraint_FAULT};
    if c == Constraint_FAULT then
        boolean secondstage = FALSE; boolean secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
if
    AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
    memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc); (acctype) && memaddrdesc.memattr
    _MemTag[memaddrdesc] = value;

```

Library pseudocode for aarch64/functions/memory/AArch64.PhysicalTag

```

// AArch64.PhysicalTag()
// =====
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
    return vaddr<59:56>;

```

Library pseudocode for aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess

```
// AArch64.TranslateAddressForAtomicAccess()
// =====
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.

AddressDescriptor AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits)
{
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;

    assert size IN {1, 2, 4, 8, 16};

    aligned = AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits);
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;

    assert size IN {1, 2, 4, 8, 16};

    aligned = AArch64.CheckAlignment(address, size, AccType_ATOMICRW, iswrite);

    // MMU or MPU lookup
    memaddrdesc = memaddrdesc = AArch64.TranslateAddress(address, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH thenthen
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if HaveMTE2Ext() && AArch64.AccessIsTagChecked(address, AccType_ATOMICRW) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then(address);
        if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, AccType_ATOMICRW, iswrite);

    return memaddrdesc;
```

Library pseudocode for aarch64/functions/memory/AddressSupportsLS64

```
// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.
boolean AddressSupportsLS64(bits(64) address);
```

Library pseudocode for aarch64/functions/memory/CheckAllInAlignedQuantity

```
// CheckAllInAlignedQuantity()
// =====
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

boolean CheckAllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    assert(size <= alignment);
    return Align(address+size-1, alignment) == Align(address, alignment);
```


Library pseudocode for aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// =====
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR[][SA0] != '0');
    else
        stack_align_check = (SCTLR[][SA] != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```



```

// CheckSingleAccessAttributes()
// =====
//
// When FEAT_LSE2 is implemented, a MemSingle[] access needs to be further assessed once the memory
// attributes are determined.
// If it was aligned to access size or targets Normal Inner Write-Back, Outer Write-Back Cacheable
// memory then it is single copy atomic and there is no alignment fault.
// If not, for exclusives, atomics and non atomic acquire release instructions - it is CONSTRAINED UNPRED
// if they generate an alignment fault. If they do not generate an alignment fault - they are
// single copy atomic.
// Otherwise it is IMPLEMENTATION DEFINED - if they are single copy atomic.
//
// The function returns (atomic, splitpair), where
// atomic indicates if the access is single copy atomic.
// splitpair indicates that a load/store pair is split into 2 single copy atomic accesses.
// when atomic and splitpair are both FALSE - the access is not single copy atomic and may be treated
// as byte accesses.

(boolean, boolean) CheckSingleAccessAttributes(bits(64) address, MemoryAttributes memattrs, integer size,
        AccType acctype, boolean iswrite, boolean aligned, boolean ispair)
acctype, boolean iswrite, boolean wasaligned, boolean ispair)
    isnormalwb = (memattrs.memtype == MemType_Normal &&
        memattrs.inner.attrs == MemAttr_WB &&
        memattrs.outer.attrs == MemAttr_WB);

    atomic = TRUE;
    splitpair = FALSE;
    if isnormalwb then return (atomic, splitpair);

    accatomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
        AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD};
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC};

    if !aligned && (accatomic || ordered) then
    if !wasaligned && (accatomic || ordered) then
        atomic = ConstrainUnpredictableBool(Unpredictable_MISALIGNEDATOMIC);
        if !atomic then
            secondstage = FALSE;
            AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
        else
            return (atomic, splitpair);

    if ispair && wasaligned then
        // load / store pair requests that are aligned to each register access are split into 2 single copy atomic
        atomic = FALSE;
        splitpair = TRUE;
        return (atomic, splitpair);

    if wasaligned then
        return (atomic, splitpair);

    atomic = boolean IMPLEMENTATION_DEFINED "Misaligned accesses within 16 byte aligned memory but not No
    return (atomic, splitpair);
    AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
    else
        return (atomic, splitpair);

    if ispair && aligned then
        // load / store pair requests that are aligned to each register access are split into 2 single copy atomic
        atomic = FALSE;
        splitpair = TRUE;
        return (atomic, splitpair);

    if aligned then
        return (atomic, splitpair);

    atomic = boolean IMPLEMENTATION_DEFINED "Misaligned accesses within 16 byte aligned memory but not No
    return (atomic, splitpair);

```

Library pseudocode for aarch64/functions/memory/IsTagCheckedInstruction

```
// Returns True if the current instruction uses tag-checked memory access,  
// False otherwise.  
boolean IsTagCheckedInstruction();
```



```

// Mem[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem(bits(64) address, integer size, AccType acctype]
    boolean ispair = FALSE;
    return Mem[address, size, acctype, ispair];

bits(size*8) Mem(bits(64) address, integer size, AccType acctype, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    boolean iswrite = FALSE;
    integer halfsize = size DIV 2;

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
    elseif acctype IN {AccType_VEC, AccType_VECSTREAM} then
        // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);
    else
        // 16-byte integer access
        atomic = address == Align(address, 16);

    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_pair = FALSE;
        single_is_aligned = TRUE;
        value1 = AArch64.MemSingle[address, halfsize, acctype, single_is_aligned, single_is_pair];
        value2 = AArch64.MemSingle[address + halfsize, halfsize, acctype, single_is_aligned, single_is_pair];
        value = value2<8*(size DIV 2)-1:0>:value1<8*(size DIV 2)-1:0>;
    elseif atomic && ispair then
        value = AArch64.MemSingle[address, size, acctype, aligned, ispair];
    elseif !atomic then

        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

            for i = 1 to size-1
                value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
        elseif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
            value<63:0> = AArch64.MemSingle[address, 8, acctype, aligned, ispair];
            value<127:64> = AArch64.MemSingle[address+8, 8, acctype, aligned, ispair];
        else
            value = AArch64.MemSingle[address, size, acctype, aligned, ispair];

        if BigEndian(acctype) then
            value = BigEndianReverse(value);

    return value;

// Mem[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

```

```

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
    boolean ispair = FALSE;
    Mem[address, size, acctype, ispair] = value;

Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value
    boolean iswrite = TRUE;
    integer halfsize = size DIV 2;

    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if ispair then
        atomic = CheckAllInAlignedQuantity(address, size, 16);
    elseif size != 16 || !(acctype IN {AccType\_VEC, AccType\_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
    elseif (acctype IN {AccType\_VEC, AccType\_VECSTREAM}) then
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);
    else
        // 16-byte integer access
        atomic = address == Align(address, 16);

    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_aligned = TRUE;
        bits(halfsize*8) val1 = value<(8*halfsize)-1:0>;
        bits(halfsize*8) val2 = value<(16*halfsize)-1:(8*halfsize)>;
        AArch64.MemSingle[address, halfsize, acctype, single_is_aligned, ispair] = val1;
        AArch64.MemSingle[address + halfsize, halfsize, acctype, single_is_aligned, ispair] = val2;
    elseif atomic && ispair then
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    elseif !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
            assert c IN {Constraint\_FAULT, Constraint\_NONE};
            if c == Constraint\_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    elseif size == 16 && acctype IN {AccType\_VEC, AccType\_VECSTREAM} then
        AArch64.MemSingle[address, 8, acctype, aligned, ispair] = value<63:0>;
        AArch64.MemSingle[address+8, 8, acctype, aligned, ispair] = value<127:64>;
    else
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;

```

Library pseudocode for aarch64/functions/memory/MemAtomic

```
// MemAtomic()
// =====
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, MemAtomicOp op, bits(size) value, AccType ldacctype, AccType stacctype,
    bits(size) newvalue;
    memaddrdesc = memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    (memstatus, oldvalue) = oldvalue = Mem[memaddrdesc, size DIV 8, ldaccdesc, FALSE];
    if PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
    if BigEndian(ldacctype) then
        oldvalue = BigEndianReverse(oldvalue);

    case op of
        when MemAtomicOp_ADD      newvalue = oldvalue + value;
        when MemAtomicOp_BIC      newvalue = oldvalue AND NOT(value);
        when MemAtomicOp_EOR      newvalue = oldvalue EOR value;
        when MemAtomicOp_ORR      newvalue = oldvalue OR value;
        when MemAtomicOp_SMAX     newvalue = if SInt(oldvalue) > SInt(value) then oldvalue else value;
        when MemAtomicOp_SMIN     newvalue = if SInt(oldvalue) > SInt(value) then value else oldvalue;
        when MemAtomicOp_UMAX     newvalue = if UInt(oldvalue) > UInt(value) then oldvalue else value;
        when MemAtomicOp_UMIN     newvalue = if UInt(oldvalue) > UInt(value) then value else oldvalue;
        when MemAtomicOp_SWP      newvalue = value;

    if BigEndian(stacctype) then
        newvalue = BigEndianReverse(newvalue);
    memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);
    (newvalue);
    _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;

    // Load operations return the old (pre-operation) value
    return oldvalue;
```


Library pseudocode for aarch64/functions/memory/MemAtomicCompareAndSwap

```
// MemAtomicCompareAndSwap()
// =====
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(size) MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
                                   bits(size) newvalue, AccType ldacctype, AccType stacctype)
    memaddrdesc = memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    (memstatus, oldvalue) = oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc, FALSE];
    if PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
    if BigEndian(ldacctype) then
        oldvalue = BigEndianReverse(oldvalue);

    if oldvalue == expectedvalue then
        if BigEndian(stacctype) then
            newvalue = BigEndianReverse(newvalue);
        memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);
    (newvalue);

    _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;
    return oldvalue;
```



```

// MemLoad64B()
// =====
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccType acctype)
    bits(512) data;
    boolean iswrite = FALSE;
    constant integer size = 64;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};

        if c == Constraint_FAULT then
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            boolean secondstage = FALSE;
            boolean s2fslwalk = FALSE;
            fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fslwalk);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level
            for i = 0 to 63
                data<7+8*i : 8*i> = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fslwalk);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level
            for i = 0 to 63
                data<7+8*i : 8*i> = AArch64.MemSingle[address+8*i, 1, acctype, aligned];
            return data;

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability !=

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress,
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveMTE2Ext() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if ! if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then AArch64.CheckTag(memaddr
            AArch64.TagCheckFault(address, acctype, iswrite);

    (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
(address, acctype, iswrite);

    data = _Mem[memaddrdesc, size, accdesc, iswrite];
    return data;

```

Library pseudocode for aarch64/functions/memory/MemStore64B

```
// MemStore64B()
// =====
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.

MemStore64B(bits(64) address, bits(512) value, AccType acctype)
    boolean iswrite = TRUE;
    constant integer size = 64;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable\_LS64UNSUPPORTED);
        assert c IN {Constraint\_LIMITED\_ATOMICITY, Constraint\_FAULT};

        if c == Constraint\_FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            boolean secondstage = FALSE;
            boolean s2fslwalk = FALSE;
            fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fslwalk);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level.
            for i = 0 to 63
                AArch64.MemSingle[address+8*i, 1, acctype, aligned] = value<7+8*i : 8*i>;
    else
        -= MemStore64BWithRet(address, value, acctype); // Return status is ignored by ST64B
    return;
```

Library pseudocode for aarch64/functions/memory/MemStore64BWithRet

```
// MemStore64BWithRet()
// =====
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccType acctype)acctype)
AddressDescriptor memaddrdesc;
boolean iswrite = TRUE;
constant integer size = 64;

aligned =
AddressDescriptor memaddrdesc;
boolean iswrite = TRUE;
constant integer size = 64;

aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
memaddrdesc = memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
    return AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
    return ZeroExtend('1');

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH thenthen
    ClearExclusiveByAddress(memaddrdesc.paddress,
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);

if HaveMTE2Ext() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if ! if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) thenAArch64.CheckTag(memaddr
        AArch64.TagCheckFault(address, acctype, iswrite);
        return ZeroExtend('1');

memstatus = _Mem[memaddrdesc, size, accdesc] = value;
status = PhysMemWriteMemStore64BWithRetStatus(memaddrdesc, size, accdesc, value);
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    return memstatus.store64bstatus;();
return status;
```

Library pseudocode for aarch64/functions/memory/MemStore64BWithRetStatus

```
// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.
bits(64) MemStore64BWithRetStatus();
```

Library pseudocode for aarch64/functions/memory/NVMem

```
// NVMem[] - non-assignment form
// =====
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

bits(64) NVMem[integer offset]
    assert offset > 0;
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    return Mem[address, 8, AccType_NV2REGISTER];

// NVMem[] - assignment form
// =====
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
    assert offset > 0;
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    Mem[address, 8, AccType_NV2REGISTER] = value;
    return;
```

Library pseudocode for aarch64/functions/memory/PhysMemTagRead

```
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
(PhysMemRetStatus, bits(4)) PhysMemTagRead(AddressDescriptor desc, AccessDescriptor accdesc);
```

Library pseudocode for aarch64/functions/memory/PhysMemTagWrite

```
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
PhysMemRetStatus PhysMemTagWrite(AddressDescriptor desc, AccessDescriptor accdesc, bits(4) value);
```

Library pseudocode for aarch64/functions/memory/SetTagCheckedInstruction

```
// Flag the current instruction as using/not using memory tag checking.
SetTagCheckedInstruction(boolean checked);
```

Library pseudocode for aarch64/functions/memory/_MemTag

```
// This _MemTag[] accessor is the hardware operation which perform a single-copy atomic,  
// Allocation Tag granule aligned, memory access from the tag in PA space.  
//  
// The function address the array using desc.paddress which supplies:  
// * A 52-bit physical address  
// * A single NS bit to select between Secure and Non-secure parts of the array.  
//  
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,  
// etc and other parameters required to access the physical memory or for setting syndrome  
// register in the event of an External abort.  
bits(4) _MemTag[AddressDescriptor desc, AccessDescriptor accdesc];  
  
// This _MemTag[] accessor is the hardware operation which perform a single-copy atomic,  
// Allocation Tag granule aligned, memory access to the tag in PA space.  
//  
// The functions address the array using desc.paddress which supplies:  
// * A 52-bit physical address  
// * A single NS bit to select between Secure and Non-secure parts of the array.  
//  
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,  
// etc and other parameters required to access the physical memory or for setting syndrome  
// register in the event of an External abort.  
_MemTag[AddressDescriptor desc, AccessDescriptor accdesc] = bits(4) value;
```



```

// AddPAC()
// =====
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
    else selbit = if tbi then ptr<55> else ptr<63>;

    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

    // The pointer authentication code field takes all the available bits in between
    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

    // Check if the ptr has good extension bits and corrupt the pointer authentication code if not
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        if HaveEnhancedPAC() then
            PAC = 0x0000000000000000<63:0>;
        elseif !HaveEnhancedPAC2() then
            PAC<top_bit-1> = NOT(PAC<top_bit-1>);

    // preserve the determination between upper and lower address at bit<55> and insert PAC

```

```

if !HaveEnhancedPAC2() then
  if tbi then
    result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
  else
    result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
else
  if tbi then
    result = ptr<63:56>:selbit:(ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
  else
    result = (ptr<63:56> EOR PAC<63:56>):selbit:(ptr<54:bottom_PAC_bit> EOR
      PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
return result;

```

Library pseudocode for aarch64/functions/pac/addpacda/AddPACDA

```

// AddPACDA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) X, bits(64) Y)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1) Enable;
  bits(128) APDAKey_EL1;

  APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
  case PSTATE.EL of
    when EL0
      boolean IsEL1Regime = S1TranslationRegime() == EL1;
      Enable = if IsEL1Regime then SCTLRL_EL1.EnDA else SCTLRL_EL2.EnDA;
      TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
      Enable = SCTLRL_EL1.EnDA;
      TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
      Enable = SCTLRL_EL2.EnDA;
      TrapEL2 = FALSE;
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
      Enable = SCTLRL_EL3.EnDA;
      TrapEL2 = FALSE;
      TrapEL3 = FALSE;

  if Enable == '0' then return X;
  elsif TrapEL2 then TrapPACUse(EL2);
  elsif TrapEL3 then TrapPACUse(EL3);
  else return AddPAC(X, Y, APDAKey_EL1, TRUE);

```

Library pseudocode for aarch64/functions/pac/addpacdb/AddPACDB

```
// AddPACDB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnDB else SCTLRL_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APDBKey_EL1, TRUE);
```

Library pseudocode for aarch64/functions/pac/addpacga/AddPACGA

```
// AddPACGA()
// =====
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of X, Y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;

    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return ComputePAC(X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0><63:32>:Zeros(32));
```

Library pseudocode for aarch64/functions/pac/addpacia/AddPACIA

```
// AddPACIA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIA else SCTLRL_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APIAKey_EL1, FALSE);
```

Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB

```
// AddPACIB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIB else SCTLRL_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APIBKey_EL1, FALSE);
```

Library pseudocode for aarch64/functions/pac/auth/AArch64.PACFailException

```
// AArch64.PACFailException()
// =====
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PACFail);
    exception.syndrome<1:0> = syndrome;
    exception.syndrome<24:2> = Zeros(); // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/pac/auth/Auth

```
// Auth()
// =====
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number, boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    if tbi then
        original_ptr = ptr<63:56>:extfield<56-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<64-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63:55>:error_code:original_ptr<52:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    else
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63>:error_code:original_ptr<60:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            result<63:56> = result<63:56> EOR PAC<63:56>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);

    return result;
```

Library pseudocode for aarch64/functions/pac/authda/AuthDA

```
// AuthDA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnDA else SCTLRL_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return Auth(X, Y, APDAKey_EL1, TRUE, '0', is_combined);
```


Library pseudocode for aarch64/functions/pac/authdb/AuthDB

```
// AuthDB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnDB else SCTLRL_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return Auth(X, Y, APDBKey_EL1, TRUE, '1', is_combined);
```

Library pseudocode for aarch64/functions/pac/authia/AuthIA

```
// AuthIA()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIA else SCTLRL_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return Auth(X, Y, APIAKey_EL1, FALSE, '0', is_combined);
```

Library pseudocode for aarch64/functions/pac/authib/AuthIB

```
// AuthIB()
// =====
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIB else SCTLRL_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return Auth(X, Y, APIBKey_EL1, FALSE, '1', is_combined);
```

Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =====

integer CalculateBottomPACBit(bit top_bit)
    integer tsz_field;

    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
            using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
        else
            // EL2 translation regime registers
            assert HaveEL(EL2);
            tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
            using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
    else
        tsz_field = if PSTATE.EL == EL2 then UInt(TCR_EL2.T0SZ) else UInt(TCR_EL3.T0SZ);
        using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';

    max_limit_tsz_field = (if !HaveSmallTranslationTableExt() then 39 else if using64k then 47 else 48);
    if tsz_field > max_limit_tsz_field then
        // TCR_ELx.TySZ is out of range
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
    tszmin = if using64k && AArch64.VAMax() == 52 then 12 else 16;
    if tsz_field < tszmin then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = tszmin;
    return (64-tsz_field);
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```
// ComputePAC()
// =====

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
    bits(64) workingval;
    bits(64) runningmod;
    bits(64) roundkey;
    bits(64) modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;
    RC[3] = 0x082EFA98EC4E6C89<63:0>;
    RC[4] = 0x452821E638D01377<63:0>;

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;
    for i = 0 to 4
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
            workingval = PACSub(workingval);
            runningmod = TweakShuffle(runningmod<63:0>);
        roundkey = modk0 EOR runningmod;
        workingval = workingval EOR roundkey;
        workingval = PACCellShuffle(workingval);
        workingval = PACMult(workingval);
        workingval = PACSub(workingval);
        workingval = PACCellShuffle(workingval);
        workingval = PACMult(workingval);
        workingval = key1 EOR workingval;
        workingval = PACCellInvShuffle(workingval);
        workingval = PACInvSub(workingval);
        workingval = PACMult(workingval);
        workingval = PACCellInvShuffle(workingval);
        workingval = workingval EOR key0;
        workingval = workingval EOR runningmod;
    for i = 0 to 4
        workingval = PACInvSub(workingval);
        if i < 4 then
            workingval = PACMult(workingval);
            workingval = PACCellInvShuffle(workingval);
            runningmod = TweakInvShuffle(runningmod<63:0>);
            roundkey = key1 EOR runningmod;
            workingval = workingval EOR RC[4-i];
            workingval = workingval EOR roundkey;
            workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```
// PACCellInvShuffle()
// =====

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

```
// PACCellShuffle()
// =====

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// =====

bits(64) PACInvSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '0101';
        when '0001' Toutput<4*i+3:4*i> = '1110';
        when '0010' Toutput<4*i+3:4*i> = '1101';
        when '0011' Toutput<4*i+3:4*i> = '1000';
        when '0100' Toutput<4*i+3:4*i> = '1010';
        when '0101' Toutput<4*i+3:4*i> = '1011';
        when '0110' Toutput<4*i+3:4*i> = '0001';
        when '0111' Toutput<4*i+3:4*i> = '1001';
        when '1000' Toutput<4*i+3:4*i> = '0010';
        when '1001' Toutput<4*i+3:4*i> = '0110';
        when '1010' Toutput<4*i+3:4*i> = '1111';
        when '1011' Toutput<4*i+3:4*i> = '0000';
        when '1100' Toutput<4*i+3:4*i> = '0100';
        when '1101' Toutput<4*i+3:4*i> = '1100';
        when '1110' Toutput<4*i+3:4*i> = '0111';
        when '1111' Toutput<4*i+3:4*i> = '0011';
return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =====

bits(64) PACMult(bits(64) Sinput)
bits(4) t0;
bits(4) t1;
bits(4) t2;
bits(4) t3;
bits(64) Soutput;

for i = 0 to 3
    t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
    t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
    t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
    t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
    t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
    t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
    Soutput<4*i+3:4*i> = t3<3:0>;
    Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
    Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
    Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
return Soutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// =====

bits(64) PACSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '1011';
        when '0001' Toutput<4*i+3:4*i> = '0110';
        when '0010' Toutput<4*i+3:4*i> = '1000';
        when '0011' Toutput<4*i+3:4*i> = '1111';
        when '0100' Toutput<4*i+3:4*i> = '1100';
        when '0101' Toutput<4*i+3:4*i> = '0000';
        when '0110' Toutput<4*i+3:4*i> = '1001';
        when '0111' Toutput<4*i+3:4*i> = '1110';
        when '1000' Toutput<4*i+3:4*i> = '0011';
        when '1001' Toutput<4*i+3:4*i> = '0111';
        when '1010' Toutput<4*i+3:4*i> = '0100';
        when '1011' Toutput<4*i+3:4*i> = '0101';
        when '1100' Toutput<4*i+3:4*i> = '1101';
        when '1101' Toutput<4*i+3:4*i> = '0010';
        when '1110' Toutput<4*i+3:4*i> = '0001';
        when '1111' Toutput<4*i+3:4*i> = '1010';
return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/RC

```
array bits(64) RC[0..4];
```

Library pseudocode for aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =====

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =====

bits(4) TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```


Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// =====

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =====

bits(64) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// =====

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/bits(64)

```
array bits(64) RC[0..4];
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC

```
// HaveEnhancedPAC()
// =====
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.

boolean HaveEnhancedPAC()
    return ( HavePACExt\(\)
        && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality" );
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC2

```
// HaveEnhancedPAC2()
// =====
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
    return HasArchVersion\(ARMv8p6\) || ( HasArchVersion\(ARMv8p3\) && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC2 functionality" );
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPAC

```
// HaveFPAC()
// =====
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.

boolean HaveFPAC()
    return HaveEnhancedPAC2\(\) && boolean IMPLEMENTATION_DEFINED "Has FPAC functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPACCombined

```
// HaveFPACCombined()
// =====
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HaveFPACCombined()
    return HaveFPAC\(\) && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACExt

```
// HavePACExt()
// =====
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACExt()
    return HasArchVersion\(ARMv8p3\);
```

Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// =====
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    regime = TranslationRegime(PSTATE.EL, AccType\_NORMAL);
    return HasUnprivileged(regime);
```

Library pseudocode for aarch64/functions/pac/strip/Strip

```
// Strip()
// =====
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);

    if tbi then
        original_ptr = A<63:56>:extfield< 56-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield< 64-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;

    return original_ptr;
```

Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// =====
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception\_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

```
// AArch64.ESBOperation()
// =====
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;

    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, EL1};
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = PSTATE.EL IN {EL0, EL2};
    else
        mask_active = PSTATE.EL == target;

    mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
                                     PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));
    intdis = Halted() || ExternalDebugInterruptsDisabled(target);
    masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            syndrome32 = AArch32.PhysicalSErrorSyndrome();
            DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        else
            implicit_esb = FALSE;
            syndrome64 = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = AArch64.ReportDeferredSError(syndrome64);
            ClearPendingPhysicalSError(); // Set ISR_EL1.A to 0

    return;
```

Library pseudocode for aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

```
// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);
```

Library pseudocode for aarch64/functions/ras/AArch64.ReportDeferredSError

```
// AArch64.ReportDeferredSError()
// =====
// Generate deferred SError syndrome

bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31> = '1'; // A
    target<24> = syndrome<24>; // IDS
    target<23:0> = syndrome<23:0>; // ISS
    return target;
```

Library pseudocode for aarch64/functions/ras/AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =====
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if ELUsingAArch32(EL1) then
            VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        else
            VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>);
            HCR_EL2.VSE = '0'; // Clear pending virtual SError

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```
// AArch64.MaybeZeroRegisterUppers()
// =====
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32(); // Always called from AArch32 state before entering AArch64 state

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0; last = 14; include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0; last = 30; include_R15 = FALSE;
    else
        first = 0; last = 30; include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[n]<63:32> = Zeros();

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// =====

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i] = bits(64) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// =====

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// =====

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(64) UNKNOWN;
        ELR_EL3 = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq<31:0> = bits(32) UNKNOWN;
        SPSR_irq<31:0> = bits(32) UNKNOWN;
        SPSR_abt<31:0> = bits(32) UNKNOWN;
        SPSR_und<31:0> = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSystemRegisters

```
AArch64.ResetSystemRegisters(boolean cold_reset);
```

Library pseudocode for aarch64/functions/registers/PC

```
// PC - non-assignment form
// =====
// Read program counter.

bits(64) PC[]
    return _PC;
```

Library pseudocode for aarch64/functions/registers/SP

```
// SP[] - assignment form
// =====
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
    assert width IN {32,64};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value);
    else
        case PSTATE.EL of
            when EL0 SP_EL0 = ZeroExtend(value);
            when EL1 SP_EL1 = ZeroExtend(value);
            when EL2 SP_EL2 = ZeroExtend(value);
            when EL3 SP_EL3 = ZeroExtend(value);
        return;

// SP[] - non-assignment form
// =====
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
    assert width IN {8,16,32,64};
    if PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0 return SP_EL0<width-1:0>;
            when EL1 return SP_EL1<width-1:0>;
            when EL2 return SP_EL2<width-1:0>;
            when EL3 return SP_EL3<width-1:0>;
```

Library pseudocode for aarch64/functions/registers/V

```
// V[] - assignment form
// =====
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    integer vlen = if IsSVEEnabled(PSTATE.EL) then VL else 128;
    if ConstrainUnpredictableBool(Unpredictable_SVEZERoupper) then
        _Z[n] = ZeroExtend(value);
    else
        _Z[n]<vlen-1:0> = ZeroExtend(value);

// V[] - non-assignment form
// =====
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _Z[n]<width-1:0>;
```

Library pseudocode for aarch64/functions/registers/Vpart

```
// Vpart[] - non-assignment form
// =====
// Reads a 128-bit SIMD&FP register in up to two parts:
// part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
// part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
// value held in the register.

bits(width) Vpart[integer n, integer part]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        return V[n];
    else
        assert width IN {32,64};
        bits(128) vreg = V[n];
        return vreg<(width * 2)-1:width>;

// Vpart[] - assignment form
// =====
// Writes a 128-bit SIMD&FP register in up to two parts:
// part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
// part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        V[n] = value;
    else
        assert width == 64;
        bits(64) vreg = V[n];
        V[n] = value<63:0> : vreg;
```

Library pseudocode for aarch64/functions/registers/X

```
// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value.

X[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value);
    return;

// X[] - non-assignment form
// =====
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
```


Library pseudocode for aarch64/functions/sve/AArch32.IsFPEnabled

```
// AArch32.IsFPEnabled()
// =====
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
    if el == EL0 && !ELUsingAArch32\(EL1\) then
        return AArch64.IsFPEnabled(el);

    if HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && !IsSecure\(\) then
        // Check if access disabled in NSACR
        if NSACR.cp10 == '0' then return FALSE;

    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        case CPACR.cp10 of
            when '00' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '10' disabled = ConstrainUnpredictableBool\(Unpredictable\_RESCPACR\);
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    if el IN {EL0, EL1, EL2} && EL2Enabled\(\) then
        if !ELUsingAArch32\(EL2\) then
            return AArch64.IsFPEnabled\(EL2\);
        if HCPTR.TCP10 == '1' then return FALSE;

    if HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/AArch64.IsFPEnabled

```
// AArch64.IsFPEnabled()
// =====
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

boolean AArch64.IsFPEnabled(bits(2) el)
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost\(\) then
        // Check FP&SIMD at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled\(\) then
        if HaveVirtHostExt\(\) && HCR_EL2.E2H == '1' then
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TFP == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL\(EL3\) then
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/AnyActiveElement

```
// AnyActiveElement()
// =====
// Return TRUE if there is at least one active element in mask. Otherwise,
// return FALSE.

boolean AnyActiveElement(bits(N) mask, integer esize)
    return LastActiveElement(mask, esize) >= 0;
```

Library pseudocode for aarch64/functions/sve/CeilPow2

```
// CeilPow2()
// =====

// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 2;
    return FloorPow2(x - 1) * 2;
```

Library pseudocode for aarch64/functions/sve/CheckSVEEnabled

```
// CheckSVEEnabled()
// =====
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

CheckSVEEnabled()
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    // Check if access disabled in CPTR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SVE at EL2
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SVEAccessTrap(EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then SVEAccessTrap(EL3);
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
```

Library pseudocode for aarch64/functions/sve/DecodePredCount

```
// DecodePredCount()
// =====

integer DecodePredCount(bits(5) pattern, integer esize)
  integer elements = VL DIV esize;
  integer numElem;
  case pattern of
    when '00000' numElem = FloorPow2(elements);
    when '00001' numElem = if elements >= 1 then 1 else 0;
    when '00010' numElem = if elements >= 2 then 2 else 0;
    when '00011' numElem = if elements >= 3 then 3 else 0;
    when '00100' numElem = if elements >= 4 then 4 else 0;
    when '00101' numElem = if elements >= 5 then 5 else 0;
    when '00110' numElem = if elements >= 6 then 6 else 0;
    when '00111' numElem = if elements >= 7 then 7 else 0;
    when '01000' numElem = if elements >= 8 then 8 else 0;
    when '01001' numElem = if elements >= 16 then 16 else 0;
    when '01010' numElem = if elements >= 32 then 32 else 0;
    when '01011' numElem = if elements >= 64 then 64 else 0;
    when '01100' numElem = if elements >= 128 then 128 else 0;
    when '01101' numElem = if elements >= 256 then 256 else 0;
    when '11101' numElem = elements - (elements MOD 4);
    when '11110' numElem = elements - (elements MOD 3);
    when '11111' numElem = elements;
    otherwise    numElem = 0;
  return numElem;
```

Library pseudocode for aarch64/functions/sve/ElemFFR

```
// ElemFFR[] - non-assignment form
// =====

bit ElemFFR[integer e, integer esize]
  return ElemP[_FFR, e, esize];

// ElemFFR[] - assignment form
// =====

ElemFFR[integer e, integer esize] = bit value
  integer psize = esize DIV 8;
  integer n = e * psize;
  assert n >= 0 && (n + psize) <= PL;
  _FFR<n+psize-1:n> = ZeroExtend(value, psize);
  return;
```

Library pseudocode for aarch64/functions/sve/ElemP

```
// ElemP[] - non-assignment form
// =====

bit ElemP[bits(N) pred, integer e, integer esize]
  integer n = e * (esize DIV 8);
  assert n >= 0 && n < N;
  return pred<n>;

// ElemP[] - assignment form
// =====

ElemP[bits(N) &pred, integer e, integer esize] = bit value
  integer psize = esize DIV 8;
  integer n = e * psize;
  assert n >= 0 && (n + psize) <= N;
  pred<n+psize-1:n> = ZeroExtend(value, psize);
  return;
```

Library pseudocode for aarch64/functions/sve/FFR

```
// FFR[] - non-assignment form
// =====

bits(width) FFR[]
    assert width == PL;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// =====

FFR[] = bits(width) value
    assert width == PL;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
        _FFR = ZeroExtend(value);
    else
        _FFR<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sve/FPCompareNE

```
// FPCompareNE()
// =====

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUntpack(op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op2, fpcr);
    op1_nan = type1 IN {FPTType\_SNaN, FPTType\_QNaN};
    op2_nan = type2 IN {FPTType\_SNaN, FPTType\_QNaN};

    if op1_nan || op2_nan then
        result = TRUE;
        if type1 == FPTType\_SNaN || type2 == FPTType\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by FPUntpack()
        result = (value1 != value2);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPCompareUN

```
// FPCompareUN()
// =====

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUntpack(op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op2, fpcr);

    if type1 == FPTType\_SNaN || type2 == FPTType\_SNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);

    result = type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN};

    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPConvertSVE

```
// FPConvertSVE()
// =====

bits(M) FPConvertSVE(bits(N) op, FPCRTType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding);

// FPConvertSVE()
// =====

bits(M) FPConvertSVE(bits(N) op, FPCRTType fpcr)
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

Library pseudocode for aarch64/functions/sve/FPExpA

```
// FPExpA()
// =====

bits(N) FPExpA(bits(N) op)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx];
    if N == 16 then
        result<15:0> = '0':op<9:5>:coeff<9:0>;
    elsif N == 32 then
        result<31:0> = '0':op<13:6>:coeff<22:0>;
    else // N == 64
        result<63:0> = '0':op<16:6>:coeff<51:0>;

    return result;
```



```

// FPExpCoefficient()
// =====

bits(N) FPExpCoefficient[integer index]
  assert N IN {16,32,64};
  integer result;

  if N == 16 then
    case index of
      when 0 result = 0x0000;
      when 1 result = 0x0016;
      when 2 result = 0x002d;
      when 3 result = 0x0045;
      when 4 result = 0x005d;
      when 5 result = 0x0075;
      when 6 result = 0x008e;
      when 7 result = 0x00a8;
      when 8 result = 0x00c2;
      when 9 result = 0x00dc;
      when 10 result = 0x00f8;
      when 11 result = 0x0114;
      when 12 result = 0x0130;
      when 13 result = 0x014d;
      when 14 result = 0x016b;
      when 15 result = 0x0189;
      when 16 result = 0x01a8;
      when 17 result = 0x01c8;
      when 18 result = 0x01e8;
      when 19 result = 0x0209;
      when 20 result = 0x022b;
      when 21 result = 0x024e;
      when 22 result = 0x0271;
      when 23 result = 0x0295;
      when 24 result = 0x02ba;
      when 25 result = 0x02e0;
      when 26 result = 0x0306;
      when 27 result = 0x032e;
      when 28 result = 0x0356;
      when 29 result = 0x037f;
      when 30 result = 0x03a9;
      when 31 result = 0x03d4;

    elsif N == 32 then
      case index of
        when 0 result = 0x000000;
        when 1 result = 0x0164d2;
        when 2 result = 0x02cd87;
        when 3 result = 0x043a29;
        when 4 result = 0x05aac3;
        when 5 result = 0x071f62;
        when 6 result = 0x08980f;
        when 7 result = 0x0a14d5;
        when 8 result = 0x0b95c2;
        when 9 result = 0x0d1adf;
        when 10 result = 0x0ea43a;
        when 11 result = 0x1031dc;
        when 12 result = 0x11c3d3;
        when 13 result = 0x135a2b;
        when 14 result = 0x14f4f0;
        when 15 result = 0x16942d;
        when 16 result = 0x1837f0;
        when 17 result = 0x19e046;
        when 18 result = 0x1b8d3a;
        when 19 result = 0x1d3eda;
        when 20 result = 0x1ef532;
        when 21 result = 0x20b051;
        when 22 result = 0x227043;
        when 23 result = 0x243516;
        when 24 result = 0x25fed7;
        when 25 result = 0x27cd94;

```



```

when 26 result = 0x29a15b;
when 27 result = 0x2b7a3a;
when 28 result = 0x2d583f;
when 29 result = 0x2f3b79;
when 30 result = 0x3123f6;
when 31 result = 0x3311c4;
when 32 result = 0x3504f3;
when 33 result = 0x36fd92;
when 34 result = 0x38fbaf;
when 35 result = 0x3aff5b;
when 36 result = 0x3d08a4;
when 37 result = 0x3f179a;
when 38 result = 0x412c4d;
when 39 result = 0x4346cd;
when 40 result = 0x45672a;
when 41 result = 0x478d75;
when 42 result = 0x49b9be;
when 43 result = 0x4bec15;
when 44 result = 0x4e248c;
when 45 result = 0x506334;
when 46 result = 0x52a81e;
when 47 result = 0x54f35b;
when 48 result = 0x5744fd;
when 49 result = 0x599d16;
when 50 result = 0x5bfbb8;
when 51 result = 0x5e60f5;
when 52 result = 0x60ccdf;
when 53 result = 0x633f89;
when 54 result = 0x65b907;
when 55 result = 0x68396a;
when 56 result = 0x6ac0c7;
when 57 result = 0x6d4f30;
when 58 result = 0x6fe4ba;
when 59 result = 0x728177;
when 60 result = 0x75257d;
when 61 result = 0x77d0df;
when 62 result = 0x7a83b3;
when 63 result = 0x7d3e0c;

```

```

else // N == 64

```

```

    case index of

```

```

        when 0 result = 0x00000000000000;
        when 1 result = 0x02C9A3E778061;
        when 2 result = 0x059B0D3158574;
        when 3 result = 0x0874518759BC8;
        when 4 result = 0x0B5586CF9890F;
        when 5 result = 0x0E3EC32D3D1A2;
        when 6 result = 0x11301D0125B51;
        when 7 result = 0x1429AAEA92DE0;
        when 8 result = 0x172B83C7D517B;
        when 9 result = 0x1A35BEB6FCB75;
        when 10 result = 0x1D4873168B9AA;
        when 11 result = 0x2063B88628CD6;
        when 12 result = 0x2387A6E756238;
        when 13 result = 0x26B4565E27CDD;
        when 14 result = 0x29E9DF51FDEE1;
        when 15 result = 0x2D285A6E4030B;
        when 16 result = 0x306FE0A31B715;
        when 17 result = 0x33C08B26416FF;
        when 18 result = 0x371A7373AA9CB;
        when 19 result = 0x3A7DB34E59FF7;
        when 20 result = 0x3DEA64C123422;
        when 21 result = 0x4160A21F72E2A;
        when 22 result = 0x44E086061892D;
        when 23 result = 0x486A2B5C13CD0;
        when 24 result = 0x4BFDAD5362A27;
        when 25 result = 0x4F9B2769D2CA7;
        when 26 result = 0x5342B569D4F82;
        when 27 result = 0x56F4736B527DA;
        when 28 result = 0x5AB07DD485429;

```

```

when 29 result = 0x5E76F15AD2148;
when 30 result = 0x6247EB03A5585;
when 31 result = 0x6623882552225;
when 32 result = 0x6A09E667F3BCD;
when 33 result = 0x6DFB23C651A2F;
when 34 result = 0x71F75E8EC5F74;
when 35 result = 0x75FEB564267C9;
when 36 result = 0x7A11473EB0187;
when 37 result = 0x7E2F336CF4E62;
when 38 result = 0x82589994CCE13;
when 39 result = 0x868D99B4492ED;
when 40 result = 0x8ACE5422AA0DB;
when 41 result = 0x8F1AE99157736;
when 42 result = 0x93737B0CDC5E5;
when 43 result = 0x97D829FDE4E50;
when 44 result = 0x9C49182A3F090;
when 45 result = 0xA0C667B5DE565;
when 46 result = 0xA5503B23E255D;
when 47 result = 0xA9E6B5579FDBF;
when 48 result = 0xAE89F995AD3AD;
when 49 result = 0xB33A2B84F15FB;
when 50 result = 0xB7F76F2FB5E47;
when 51 result = 0xBCC1E904BC1D2;
when 52 result = 0xC199BDD85529C;
when 53 result = 0xC67F12E57D14B;
when 54 result = 0xCB720DCEF9069;
when 55 result = 0xD072D4A07897C;
when 56 result = 0xD5818DCFBA487;
when 57 result = 0xDA9E603DB3285;
when 58 result = 0xDFC97337B9B5F;
when 59 result = 0xE502EE78B3FF6;
when 60 result = 0xEA4AFA2A490DA;
when 61 result = 0xEFA1BEE615A27;
when 62 result = 0xF50765B6E4540;
when 63 result = 0xFA7C1819E90D8;

```

```
return result<N-1:0>;
```

Library pseudocode for aarch64/functions/sve/FPMinNormal

```

// FPMinNormal()
// =====

bits(N) FPMinNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Zeros(E-1):'1';
    frac = Zeros(F);
    return sign : exp : frac;

```

Library pseudocode for aarch64/functions/sve/FPOne

```

// FPOne()
// =====

bits(N) FPOne(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = Zeros(F);
    return sign : exp : frac;

```

Library pseudocode for aarch64/functions/sve/FPPointFive

```
// FPPointFive()
// =====

bits(N) FPPointFive(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-2):'0';
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for aarch64/functions/sve/FPProcess

```
// FPProcess()
// =====

bits(N) FPProcess(bits(N) input)
    bits(N) result;
    assert N IN {16,32,64};
    FPCRTType fpcr = FPCR[];
    (fptype,sign,value) = FPUnpack(input, fpcr);

    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, input, fpcr);
    elsif fptype == FPTType\_Infinity then
        result = FPInfinity(sign);
    elsif fptype == FPTType\_Zero then
        result = FPZero(sign);
    else
        result = FPRound(value, fpcr);

        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPScale

```
// FPScale()
// =====

bits(N) FPScale(bits (N) op, integer scale, FPCRTType fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTType\_Zero then
        result = FPZero(sign);
    elsif fptype == FPTType\_Infinity then
        result = FPInfinity(sign);
    else
        result = FPRound(value * (2.0scale), fpcr);

        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigMAdd

```
// FPTrigMAdd()
// =====

bits(N) FPTrigMAdd(integer x, bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    assert x >= 0;
    assert x < 8;
    bits(N) coeff;

    if op2<N-1> == '1' then
        x = x + 8;

    coeff = FPTrigMAddCoefficient[x];
    op2   = FPAbs(op2);
    result = FPMulAdd(coeff, op1, op2, fpcr);
    return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigMAddCoefficient

```
// FPTrigMAddCoefficient()
// =====

bits(N) FPTrigMAddCoefficient[integer index]
  assert N IN {16,32,64};
  integer result;

  if N == 16 then
    case index of
      when 0 result = 0x3c00;
      when 1 result = 0xb155;
      when 2 result = 0x2030;
      when 3 result = 0x0000;
      when 4 result = 0x0000;
      when 5 result = 0x0000;
      when 6 result = 0x0000;
      when 7 result = 0x0000;
      when 8 result = 0x3c00;
      when 9 result = 0xb800;
      when 10 result = 0x293a;
      when 11 result = 0x0000;
      when 12 result = 0x0000;
      when 13 result = 0x0000;
      when 14 result = 0x0000;
      when 15 result = 0x0000;
    elsif N == 32 then
      case index of
        when 0 result = 0x3f800000;
        when 1 result = 0xbe2aaaab;
        when 2 result = 0x3c088886;
        when 3 result = 0xb95008b9;
        when 4 result = 0x36369d6d;
        when 5 result = 0x00000000;
        when 6 result = 0x00000000;
        when 7 result = 0x00000000;
        when 8 result = 0x3f800000;
        when 9 result = 0xbf000000;
        when 10 result = 0x3d2aaaa6;
        when 11 result = 0xbab60705;
        when 12 result = 0x37cd37cc;
        when 13 result = 0x00000000;
        when 14 result = 0x00000000;
        when 15 result = 0x00000000;
      else // N == 64
        case index of
          when 0 result = 0x3ff0000000000000;
          when 1 result = 0xbfc5555555555543;
          when 2 result = 0x3f8111111110f30c;
          when 3 result = 0xbf2a01a019b92fc6;
          when 4 result = 0x3ec71de351f3d22b;
          when 5 result = 0xbe5ae5e2b60f7b91;
          when 6 result = 0x3de5d8408868552f;
          when 7 result = 0x0000000000000000;
          when 8 result = 0x3ff0000000000000;
          when 9 result = 0xbfe0000000000000;
          when 10 result = 0x3fa5555555555536;
          when 11 result = 0xbf56c16c16c13a0b;
          when 12 result = 0x3efa01a019b1e8d8;
          when 13 result = 0xbe927e4f7282f468;
          when 14 result = 0x3e21ee96d2641b13;
          when 15 result = 0xbda8f76380fbb401;

  return result<N-1:0>;
```

Library pseudocode for aarch64/functions/sve/FPTrigSMul

```
// FPTrigSMul()
// =====

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    result = FPMul(op1, op1, fpcr);
    fpexc = FALSE;
    (fptype, sign, value) = FPUntpack(result, fpcr, fpexc);

    if !(fptype IN {FPTType\_QNaN, FPTType\_SNaN}) then
        result<N-1> = op2<0>;

    return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigSSel

```
// FPTrigSSel()
// =====

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};
    bits(N) result;

    if op2<0> == '1' then
        result = FP0ne(op2<1>);
    elsif op2<1> == '1' then
        result = FPNeg(op1);
    else
        result = op1;

    return result;
```

Library pseudocode for aarch64/functions/sve/FirstActive

```
// FirstActive()
// =====

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';
```

Library pseudocode for aarch64/functions/sve/FloorPow2

```
// FloorPow2()
// =====
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
    assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);
```

Library pseudocode for aarch64/functions/sve/HaveSVE

```
// HaveSVE()
// =====

boolean HaveSVE()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Have SVE ISA";
```

Library pseudocode for aarch64/functions/sve/HaveSVEFP32MatMulExt

```
// HaveSVEFP32MatMulExt()
// =====
// Returns TRUE if single-precision floating-point matrix multiply instruction support implemented and FALSE otherwise.

boolean HaveSVEFP32MatMulExt()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP32 Matrix Multiply extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVEFP64MatMulExt

```
// HaveSVEFP64MatMulExt()
// =====
// Returns TRUE if double-precision floating-point matrix multiply instruction support implemented and FALSE otherwise.

boolean HaveSVEFP64MatMulExt()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP64 Matrix Multiply extension";
```

Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

```
// ImplementedSVEVectorLength()
// =====
// Reduce SVE vector length to a supported value (e.g. power of two)

integer ImplementedSVEVectorLength(integer nbits)
    return integer IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/functions/sve/IsEven

```
// IsEven()
// =====

boolean IsEven(integer val)
    return val MOD 2 == 0;
```

Library pseudocode for aarch64/functions/sve/IsFPEnabled

```
// IsFPEnabled()
// =====
// Returns TRUE if accesses to the Advanced SIMD and floating-point
// registers are enabled at the target exception level in the current
// execution state and FALSE otherwise.

boolean IsFPEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return AArch32.IsFPEnabled(el);
    else
        return AArch64.IsFPEnabled(el);
```

Library pseudocode for aarch64/functions/sve/IsSVEEnabled

```
// IsSVEEnabled()
// =====
// Returns TRUE if access to SVE instructions and System registers is
// enabled at the target exception level and FALSE otherwise.

boolean IsSVEEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/LastActive

```
// LastActive()
// =====

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';
```

Library pseudocode for aarch64/functions/sve/LastActiveElement

```
// LastActiveElement()
// =====

integer LastActiveElement(bits(N) mask, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    integer elements = VL DIV esize;
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return e;
    return -1;
```


Library pseudocode for aarch64/functions/sve/MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// =====

MaybeZeroSVEUppers(bits(2) target_el)
    boolean lower_enabled;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
        return;

    if target_el == EL3 then
        if EL2Enabled() then
            lower_enabled = IsFPEnabled(EL2);
        else
            lower_enabled = IsFPEnabled(EL1);
    elsif target_el == EL2 then
        assert !ELUsingAArch32(EL2);
        if HCR_EL2.TGE == '0' then
            lower_enabled = IsFPEnabled(EL1);
        else
            lower_enabled = IsFPEnabled(EL0);
    else
        assert target_el == EL1 && !ELUsingAArch32(EL1);
        lower_enabled = IsFPEnabled(EL0);

    if lower_enabled then
        integer vl = if IsSVEEnabled(PSTATE.EL) then VL else 128;
        integer pl = vl DIV 8;
        for n = 0 to 31
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _Z[n] = ZeroExtend(_Z[n]<vl-1:0>);
        for n = 0 to 15
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _P[n] = ZeroExtend(_P[n]<pl-1:0>);
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _FFR = ZeroExtend(_FFR<pl-1:0>);
```

Library pseudocode for aarch64/functions/sve/MemNF

```
// MemNF[] - non-assignment form
// =====

(bits(8*size), boolean) MemNF(bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(8*size) value;

    aligned = (address == Align(address, size));
    A = SCTLR[].A;

    if !aligned && (A == '1') then
        return (bits(8*size) UNKNOWN, TRUE);

    atomic = aligned || size == 1;

    if !atomic then
        (value<7:0>, bad) = MemSingleNF[address, 1, acctype, aligned];

        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
            assert c IN {Constraint\_FAULT, Constraint\_NONE};
            if c == Constraint\_NONE then aligned = TRUE;

        for i = 1 to size-1
            (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, acctype, aligned];

            if bad then
                return (bits(8*size) UNKNOWN, TRUE);
    else
        (value, bad) = MemSingleNF[address, size, acctype, aligned];
        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    return (value, FALSE);
```

Library pseudocode for aarch64/functions/sve/MemSingleNF

```
// MemSingleNF[] - non-assignment form
// =====

(bits(8*size), boolean)(bits(8*size), boolean) MemSingleNF[bits(64) address, integer size, MemSingleNF[b
    assert acctype IN {bits(8*size) value;
    boolean iswrite = FALSE;
    AddressDescriptor memaddrdesc;

    // Implementation may suppress NF load for any reason
    if AccType_CNOTFIRST, AccType_NONFAULT};
    bits(8*size) value;
    boolean iswrite = FALSE;
    AddressDescriptor memaddrdesc;

    // Implementation may suppress NF load for any reason
    if ConstrainUnpredictableBool(Unpredictable_NONFAULT) then
        return (bits(8*size) UNKNOWN, TRUE);

    // MMU or MPU
    memaddrdesc = memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Non-fault load from Device memory must not be performed externally
    if memaddrdesc.memattrs.memtype == AArch64.TranslateAddress(address, acctype, iswrite, aligned, size)

    // Non-fault load from Device memory must not be performed externally
    if memaddrdesc.memattrs.memtype == MemType_Device then
        return (bits(8*size) UNKNOWN, TRUE);

    // Check for aborts or debug exceptions
    if if IsFault(memaddrdesc) then
        return (bits(8*size) UNKNOWN, TRUE);

    // Memory array access
    accdesc = IsFault(memaddrdesc) then
        return (bits(8*size) UNKNOWN, TRUE);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveMTE2Ext() then
        if AArch64.AccessIsTagChecked(address, acctype) then
            bits(4) ptag = AArch64.PhysicalTag(address);
            if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                return (bits(8*size) UNKNOWN, TRUE);

    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        fault = NoFault();
        fault.errortype = memstatus.errortype;
        fault.acctype = memstatus.acctype;
        fault.extflag = memstatus.extflag;
        fault.statuscode = memstatus.statuscode;
        if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc,
            size, accdesc) then
            return (bits(8*size) UNKNOWN, TRUE);
        PendSErrorInterrupt(fault);
    (address);
    if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
        return (bits(8*size) UNKNOWN, TRUE);
    value = _Mem[memaddrdesc, size, accdesc, iswrite];

    return (value, FALSE);
```

Library pseudocode for aarch64/functions/sve/NoneActive

```
// NoneActive()
// =====

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' && ElemP[x, e, esize] == '1' then return '0';
    return '1';
```

Library pseudocode for aarch64/functions/sve/P

```
// P[] - non-assignment form
// =====

bits(width) P[integer n]
    assert n >= 0 && n <= 31;
    assert width == PL;
    return _P[n]<width-1:0>;

// P[] - assignment form
// =====

P[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == PL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZERoupper) then
        _P[n] = ZeroExtend(value);
    else
        _P[n]<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sve/PL

```
// PL - non-assignment form
// =====

integer PL
    return VL DIV 8;
```

Library pseudocode for aarch64/functions/sve/PredTest

```
// PredTest()
// =====

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;
```

Library pseudocode for aarch64/functions/sve/ReducePredicated

```
// ReducePredicated()
// =====

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    integer p2bits = CeilPow2(N);
    bits(p2bits) operand;
    integer elements = p2bits DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ElemP[mask, e, esize] == '1' then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return Reduce(op, operand, esize);
```

Library pseudocode for aarch64/functions/sve/Reverse

```
// Reverse()
// =====
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    bits(N) result;
    integer sw = N DIV M;
    assert N == sw * M;
    for s = 0 to sw-1
        Elem[result, sw - 1 - s, M] = Elem[word, s, M];
    return result;
```

Library pseudocode for aarch64/functions/sve/SVEAccessTrap

```
// SVEAccessTrap()
// =====
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    exception = ExceptionSyndrome(Exception\_SVEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/sve/SVECmp

```
enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```

Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

```
// SVEMoveMaskPreferred()
// =====
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM> alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'ffffffffffffxy' or '00000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then
            return FALSE;

        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'ffxyffxyffxyffxy' or '00xy00xy00xy00xy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (IsZero(imm<15:7>) || IsOnes(imm<15:7>)) then
            return FALSE;

        // Check for 'xyxyxyxyxyxyxyxy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (imm<15:8> == imm<7:0>) then
            return FALSE;

    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffxy00' or '00000000000000xy00'
        if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then
            return FALSE;

        // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'xy00xy00xy00xy00'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
            return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/System

```
constant integer MAX_VL = 2048;
constant integer MAX_PL = 256;
array bits(MAX_VL) _Z[0..31];
array bits(MAX_PL) _P[0..15];
bits(MAX_PL) _FFR;
```

Library pseudocode for aarch64/functions/sve/VL

```
// VL - non-assignment form
// =====

integer VL
integer vl;

if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
    vl = UInt(ZCR_EL1.LEN);

if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
    vl = UInt(ZCR_EL2.LEN);
elseif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
    vl = Min(vl, UInt(ZCR_EL2.LEN));

if PSTATE.EL == EL3 then
    vl = UInt(ZCR_EL3.LEN);
elseif HaveEL(EL3) then
    vl = Min(vl, UInt(ZCR_EL3.LEN));

vl = (vl + 1) * 128;
vl = ImplementedSVEVectorLength(vl);

return vl;
```

Library pseudocode for aarch64/functions/sve/Z

```
// Z[] - non-assignment form
// =====

bits(width) Z[integer n]
    assert n >= 0 && n <= 31;
    assert width == VL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// =====

Z[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == VL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZERoupper) then
        _Z[n] = ZeroExtend(value);
    else
        _Z[n]<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTL

```
// CNTKCTL[] - non-assignment form
// =====

CNTKCTLType CNTKCTL[]
    bits(64) r;
    if IsInHost() then
        r = CNTHCTL_EL2;
        return r;
    r = CNTKCTL_EL1;
    return r;
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

Library pseudocode for aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// =====

CPACRTYPE CPACR[]
  bits(64) r;
  if IsInHost() then
    r = CPTR_EL2;
    return r;
  r = CPACR_EL1;
  return r;
```

Library pseudocode for aarch64/functions/sysregisters/CPACRTYPE

```
type CPACRTYPE;
```

Library pseudocode for aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// =====

bits(64) ELR[bits(2) el]
  bits(64) r;
  case el of
    when EL1  r = ELR_EL1;
    when EL2  r = ELR_EL2;
    when EL3  r = ELR_EL3;
    otherwise Unreachable();
  return r;

// ELR[] - non-assignment form
// =====

bits(64) ELR[]
  assert PSTATE.EL != EL0;
  return ELR[PSTATE.EL];

// ELR[] - assignment form
// =====

ELR[bits(2) el] = bits(64) value
  bits(64) r = value;
  case el of
    when EL1  ELR_EL1 = r;
    when EL2  ELR_EL2 = r;
    when EL3  ELR_EL3 = r;
    otherwise Unreachable();
  return;

// ELR[] - assignment form
// =====

ELR[] = bits(64) value
  assert PSTATE.EL != EL0;
  ELR[PSTATE.EL] = value;
  return;
```


Library pseudocode for aarch64/functions/sysregisters/ESR

```
// ESR[] - non-assignment form
// =====

ESRType ESR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1    r = ESR_EL1;
        when EL2    r = ESR_EL2;
        when EL3    r = ESR_EL3;
        otherwise Unreachable\(\);
    return r;

// ESR[] - non-assignment form
// =====

ESRType ESR[]
    return ESR\[S1TranslationRegime\\(\\)\];

// ESR[] - assignment form
// =====

ESR[bits(2) regime] = ESRType value
    bits(64) r = value;
    case regime of
        when EL1    ESR_EL1 = r;
        when EL2    ESR_EL2 = r;
        when EL3    ESR_EL3 = r;
        otherwise Unreachable\(\);
    return;

// ESR[] - assignment form
// =====

ESR[] = ESRType value
    ESR\[S1TranslationRegime\\(\\)\] = value;
```

Library pseudocode for aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

Library pseudocode for aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// =====

bits(64) FAR[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1   r = FAR_EL1;
    when EL2   r = FAR_EL2;
    when EL3   r = FAR_EL3;
    otherwise Unreachable\(\);
  return r;

// FAR[] - non-assignment form
// =====

bits(64) FAR[]
  return FAR\[S1TranslationRegime\(\)\];

// FAR[] - assignment form
// =====

FAR[bits(2) regime] = bits(64) value
  bits(64) r = value;
  case regime of
    when EL1   FAR_EL1 = r;
    when EL2   FAR_EL2 = r;
    when EL3   FAR_EL3 = r;
    otherwise Unreachable\(\);
  return;

// FAR[] - assignment form
// =====

FAR[] = bits(64) value
  FAR\[S1TranslationRegime\(\)\] = value;
  return;
```

Library pseudocode for aarch64/functions/sysregisters/MAIR

```
// MAIR[] - non-assignment form
// =====

MAIRType MAIR[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1   r = MAIR_EL1;
    when EL2   r = MAIR_EL2;
    when EL3   r = MAIR_EL3;
    otherwise Unreachable\(\);
  return r;

// MAIR[] - non-assignment form
// =====

MAIRType MAIR[]
  return MAIR\[S1TranslationRegime\(\)\];
```

Library pseudocode for aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

Library pseudocode for aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = SCTLR_EL1;
        when EL2 r = SCTLR_EL2;
        when EL3 r = SCTLR_EL3;
        otherwise Unreachable\(\);
    return r;

// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLR[]
    return SCTLR\[S1TranslationRegime\(\)\];
```

Library pseudocode for aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

Library pseudocode for aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// =====

bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = VBAR_EL1;
        when EL2 r = VBAR_EL2;
        when EL3 r = VBAR_EL3;
        otherwise Unreachable\(\);
    return r;

// VBAR[] - non-assignment form
// =====

bits(64) VBAR[]
    return VBAR\[S1TranslationRegime\(\)\];
```

Library pseudocode for aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

```
// AArch64.AllocationTagAccessIsEnabled()
// =====
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled(AccType acctype)
    bits(2) el = AArch64.AccessUsesEL(acctype);

    if SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
        return FALSE;
    elsif HCR_EL2.ATA == '0' && el IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' then
        return FALSE;
    elsif SCTLR_EL3.ATA == '0' && el == EL3 then
        return FALSE;
    elsif SCTLR_EL2.ATA == '0' && el == EL2 then
        return FALSE;
    elsif SCTLR_EL1.ATA == '0' && el == EL1 then
        return FALSE;
    elsif SCTLR_EL2.ATA0 == '0' && el == EL0 && EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
        return FALSE;
    elsif SCTLR_EL1.ATA0 == '0' && el == EL0 && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') then
        return FALSE;
    else
        return TRUE;
```

Library pseudocode for aarch64/functions/system/AArch64.ChooseNonExcludedTag

```
// AArch64.ChooseNonExcludedTag()
// =====
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag, bits(4) offset, bits(16) exclude)
    if IsOnes(exclude) then
        return '0000';

    if offset == '0000' then
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    return tag;
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingBRorBLRorRetInstr

```
// AArch64.ExecutingBRorBLRorRetInstr()
// =====
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBRorBLRorRetInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingBTIInstr

```
// AArch64.ExecutingBTIInstr()
// =====
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
        CRm = instr<11:8>;
        op2 = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingERETInstr

```
// AArch64.ExecutingERETInstr()
// =====
// Returns TRUE if current instruction is ERET.

boolean AArch64.ExecutingERETInstr()
    instr = ThisInstr();
    return instr<31:12> == '11010110100111110000';
```

Library pseudocode for aarch64/functions/system/AArch64.NextRandomTagBit

```
// AArch64.NextRandomTagBit()
// =====
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
    bits(16) lfsr = RGSr_EL1.SEED;
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RGSr_EL1.SEED = top:lfsr<15:1>;
    return top;
```

Library pseudocode for aarch64/functions/system/AArch64.RandomTag

```
// AArch64.RandomTag()
// =====
// Generate a random Allocation Tag.

bits(4) AArch64.RandomTag()
    bits(4) tag;
    for i = 0 to 3
        tag<i> = AArch64.NextRandomTagBit();
    return tag;
```

Library pseudocode for aarch64/functions/system/AArch64.SysInstr

```
// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

Library pseudocode for aarch64/functions/system/AArch64.SysInstrWithResult

```
// Execute a system instruction with read (result operand).
// Returns the result of the instruction.
bits(64) AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2);
```

Library pseudocode for aarch64/functions/system/AArch64.SysRegRead

```
// Read from a system register and return the contents of the register.  
bits(64) AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2);
```

Library pseudocode for aarch64/functions/system/AArch64.SysRegWrite

```
// Write to a system register.  
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

Library pseudocode for aarch64/functions/system/BTypeCompatible

```
boolean BTypeCompatible;
```

Library pseudocode for aarch64/functions/system/BTypeCompatible_BTI

```
// BTypeCompatible_BTI  
// =====  
// This function determines whether a given hint encoding is compatible with the current value of  
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.  
  
boolean BTypeCompatible_BTI(bits(2) hintcode)  
    case hintcode of  
        when '00'  
            return FALSE;  
        when '01'  
            return PSTATE.BTYPE != '11';  
        when '10'  
            return PSTATE.BTYPE != '10';  
        when '11'  
            return TRUE;
```

Library pseudocode for aarch64/functions/system/BTypeCompatible_PACIXSP

```
// BTypeCompatible_PACIXSP()  
// =====  
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,  
// FALSE otherwise.  
  
boolean BTypeCompatible_PACIXSP()  
    if PSTATE.BTYPE IN {'01', '10'} then  
        return TRUE;  
    elsif PSTATE.BTYPE == '11' then  
        index = if PSTATE.EL == EL0 then 35 else 36;  
        return SCTLR[]<index> == '0';  
    else  
        return FALSE;
```

Library pseudocode for aarch64/functions/system/BTypeNext

```
bits(2) BTypeNext;
```

Library pseudocode for aarch64/functions/system/ChooseRandomNonExcludedTag

```
// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random  
// Allocation Tags.  
//  
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where  
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.  
//  
// This function is permitted to generate a non-deterministic selection from the set of non-excluded  
// Allocation Tags. A reasonable implementation is described by the Pseudocode used when  
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit(). Implementations  
// may choose to behave the same as GCR_EL1.RRND=0.  
bits(4) ChooseRandomNonExcludedTag(bits(16) exclude);
```

Library pseudocode for aarch64/functions/system/InGuardedPage

```
boolean InGuardedPage;
```

Library pseudocode for aarch64/functions/system/IsHCRXEL2Enabled

```
// IsHCRXEL2Enabled()
// =====
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

boolean IsHCRXEL2Enabled()
    assert(HaveFeatHCX());
    if HaveEL(EL3) && SCR_EL3.HXEn == '0' then
        return FALSE;

    return EL2Enabled();
```

Library pseudocode for aarch64/functions/system/SetBTypeCompatible

```
// SetBTypeCompatible()
// =====
// Sets the value of BTypeCompatible global variable used by BTI

SetBTypeCompatible(boolean x)
    BTypeCompatible = x;
```

Library pseudocode for aarch64/functions/system/SetBTypeNext

```
// SetBTypeNext()
// =====
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
    BTypeNext = x;
```

Library pseudocode for aarch64/functions/system/SetInGuardedPage

```
// SetInGuardedPage()
// =====
// Global state updated to denote if memory access is from a guarded page.

SetInGuardedPage(boolean guardedpage)
    InGuardedPage = guardedpage;
```

Library pseudocode for aarch64/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =====

AArch64.ExceptionReturn(bits(64) new_pc, bits(64) spsr)

    if HaveIESB() then
        sync_errors = SCTL[0].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
        SynchronizeContext();

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    bits(2) source_el = PSTATE.EL;
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' && spsr<4> == '1' && spsr<20> == '0' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elseif UsingAArch32() then // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0'; // T32
        else
            new_pc<1:0> = '00'; // A32
    else // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc);

    if UsingAArch32() then
        // 32 most significant bits are ignored.
        boolean branch_conditional = FALSE;
        BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
    else
        BranchToAddr(new_pc, BranchType_ERET);

    CheckExceptionCatch(FALSE); // Check for debug event on exception return
```

Library pseudocode for aarch64/instrs/countop/CountOp

```
enumeration CountOp {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

Library pseudocode for aarch64/instrs/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =====
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```


Library pseudocode for aarch64/instrs/extendreg/ExtendReg

```
// ExtendReg()
// =====
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;

    case exttype of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SCTX unsigned = FALSE; len = 64;
        when ExtendType_UXTB unsigned = TRUE; len = 8;
        when ExtendType_UXTH unsigned = TRUE; len = 16;
        when ExtendType_UXTW unsigned = TRUE; len = 32;
        when ExtendType_UCTX unsigned = TRUE; len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

Library pseudocode for aarch64/instrs/extendreg/ExtendType

```
enumeration ExtendType {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SCTX,
                        ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UCTX};
```

Library pseudocode for aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```
enumeration FPMaxMinOp {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                        FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

Library pseudocode for aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp

```
enumeration FPUnaryOp {FPUnaryOp_ABS, FPUnaryOp_MOV,
                       FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

Library pseudocode for aarch64/instrs/float/convert/fpconvop/FPConvOp

```
enumeration FPConvOp {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                      FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF,
                      , FPConvOp_CVT_FtoI_JS
};
```

Library pseudocode for aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// =====
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
```



```

// DecodeBitMasks()
// =====

// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)
    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
    bits(6) tmask_or, wmask_or;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    S = UInt(imms AND levels);
    R = UInt(immr AND levels);
    diff = S - R;    // 6-bit subtract with borrow

    // From a software perspective, the remaining code is equivalent to:
    //   esize = 1 << len;
    //   d = UInt(diff<len-1:0>);
    //   welem = ZeroExtend(Ones(S + 1), esize);
    //   telem = ZeroExtend(Ones(d + 1), esize);
    //   wmask = Replicate(ROR(welem, R));
    //   tmask = Replicate(telem);
    //   return (wmask, tmask);

    // Compute "top mask"
    tmask_and = diff<5:0> OR NOT(levels);
    tmask_or = diff<5:0> AND levels;

    tmask = Ones(64);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
        OR Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
        OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
        OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
        OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
        OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
        OR Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

    // Compute "wraparound mask"
    wmask_and = immr OR NOT(levels);
    wmask_or = immr AND levels;

    wmask = Zeros(64);
    wmask = ((wmask

```

```

        AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
        OR Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
// optimization of first step:
// wmask = Replicate(wmask_or<0> : '0', 32);
wmask = ((wmask
        AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
        OR Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
wmask = ((wmask
        AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
        OR Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
        AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
        OR Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask
        AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
        OR Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
wmask = ((wmask
        AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
        OR Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

if diff<6> != '0' then // borrow from S - R
    wmask = wmask AND tmask;
else
    wmask = wmask OR tmask;

return (wmask<M-1:0>, tmask<M-1:0>);

```

Library pseudocode for aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp

```

enumeration MoveWideOp {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};

```

Library pseudocode for aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

```

// MoveWidePreferred()
// =====
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && immN:imms != 'lxxxxxx' then
        return FALSE;
    if sf == '0' && immN:imms != '00xxxxx' then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if S < 16 then
        // ones must not span halfword boundary when rotated
        return (-R MOD 16) <= (15 - S);

    // for MOVN must contain no more than 16 zeros
    if S >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (R MOD 16) <= (S - (width - 15));

    return FALSE;

```

Library pseudocode for aarch64/instrs/integer/shiftreg/DecodeShift

```
// DecodeShift()
// =====
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00' return ShiftType_LSL;
        when '01' return ShiftType_LSR;
        when '10' return ShiftType_ASR;
        when '11' return ShiftType_ROR;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftReg

```
// ShiftReg()
// =====
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shifttype, integer amount)
    bits(N) result = X[reg];
    case shifttype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftType

```
enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

Library pseudocode for aarch64/instrs/logicalop/LogicalOp

```
enumeration LogicalOp {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

Library pseudocode for aarch64/instrs/memory/memop/MemAtomicOp

```
enumeration MemAtomicOp {MemAtomicOp_ADD,
                          MemAtomicOp_BIC,
                          MemAtomicOp_EOR,
                          MemAtomicOp_ORR,
                          MemAtomicOp_SMAX,
                          MemAtomicOp_SMIN,
                          MemAtomicOp_UMAX,
                          MemAtomicOp_UMIN,
                          MemAtomicOp_SWP};
```

Library pseudocode for aarch64/instrs/memory/memop/MemOp

```
enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

Library pseudocode for aarch64/instrs/memory/prefetch/Prefetch

```
// Prefetch()
// =====

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;           // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;           // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;          // PST: prepare for store
        when '11' return;                          // unallocated hint
    target = UInt(prfop<2:1>);                      // target cache level
    stream = (prfop<0> != '0');                    // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

Library pseudocode for aarch64/instrs/system/barriers/barrierop/MemBarrierOp

```
enumeration MemBarrierOp { MemBarrierOp_DSB      // Data Synchronization Barrier
                           , MemBarrierOp_DMB      // Data Memory Barrier
                           , MemBarrierOp_ISB      // Instruction Synchronization Barrier
                           , MemBarrierOp_SSBB     // Speculative Synchronization Barrier to VA
                           , MemBarrierOp_PSSBB    // Speculative Synchronization Barrier to PA
                           , MemBarrierOp_SB       // Speculation Barrier
                           };
```

Library pseudocode for aarch64/instrs/system/hints/syshintop/SystemHintOp

```
enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI,
    SystemHintOp_WFET,
    SystemHintOp_WFIT,
    SystemHintOp_CSDB
};
```

Library pseudocode for aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                        PSTATEField_PAN, // Armv8.1
                        PSTATEField_UA0, // Armv8.2
                        PSTATEField_DIT, // Armv8.4
                        PSTATEField_SSBS,
                        PSTATEField_TC0, // Armv8.5
                        PSTATEField_SP
                        };
```

Library pseudocode for aarch64/instrs/system/sysops/dc/AArch64.DC

```
// AArch64.DC()
// =====
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope)
    AccType acctype = AccType_DC;
    CacheRecord cache;

    cache.acctype = acctype;
    cache.cachetype = cachetype;
    cache.cacheop = cacheop;
    cache.opscope = opscope;

    if opscope == CacheOpScope_SetWay then
        cache.shareability = Shareability_NSH;
        (cache.set, cache.way, cache.level) = DecodeSW(regval, cachetype);
        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            (HCR_EL2.SWI0 == '1' || HCR_EL2.<DC,VM> != '00')) then
            cache.cacheop = CacheOp_CleanInvalidate;

    CACHE_OP(cache);
    return;

    if opscope == CacheOpScope_PoDP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoDP"
        opscope = CacheOpScope_PoP;
    if opscope == CacheOpScope_PoP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoP"
        opscope = CacheOpScope_PoC;
    need_translate = DCInstNeedsTranslation(opscope);
    iswrite = cacheop == CacheOp_Invalidate;
    vaddress = regval;

    size = 0; // by default no watchpoint address
    if iswrite then
        size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
        assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
        assert (size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
        vaddress = Align(regval, size);

    cache.translated = need_translate;
    cache.vaddress = vaddress;

    if need_translate then
        wasaligned = TRUE;
        memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);
        if IsFault(memaddrdesc) then
            AArch64.Abort(regval, memaddrdesc.fault);

    memattrs = memaddrdesc.memattrs;
    cache.paddress = memaddrdesc.paddress;
    if opscope IN {CacheOpScope_PoC, CacheOpScope_PoP, CacheOpScope_PoDP} then
        cache.shareability = memattrs.shareability;
    else
        cache.shareability = Shareability_NSH;
    else
        cache.shareability = Shareability_UNKNOWN;
        cache.paddress = FullAddress_UNKNOWN;

    if cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.<DC,VM> != '00' then
        cache.cacheop = CacheOp_CleanInvalidate;

    CACHE_OP(cache);
    return;
```


Library pseudocode for aarch64/instrs/system/sysops/dc/AArch64.MemZero

```
// AArch64.MemZero()
// =====

AArch64.MemZero(bits(64) regval, CacheType cachetype)

    AccType acctype = AccType_DCZVA;
    boolean iswrite = TRUE;
    boolean wasaligned = TRUE;

    integer size = 4*(2^(UInt(DCZID_EL0.BS)));
    bits(64) vaddress = Align(regval, size);

    memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);

    if IsFault(memaddrdesc) then
        if IsDebugException(memaddrdesc.fault) then
            AArch64.Abort(vaddress, memaddrdesc.fault);
        else
            AArch64.Abort(regval, memaddrdesc.fault);
    else
        if cachetype == CacheType_Data then
            AArch64.DataMemZero(regval, vaddress, memaddrdesc, size);
        elseif cachetype == CacheType_Tag then
            if HaveMTEExt() then AArch64.TagMemZero(vaddress, size);
        elseif cachetype == CacheType_Data_Tag then
            if HaveMTEExt() then AArch64.TagMemZero(vaddress, size);
            AArch64.DataMemZero(regval, vaddress, memaddrdesc, size);

    return;
```

Library pseudocode for aarch64/instrs/system/sysops/ic/AArch64.IC

```
// AArch64.IC()
// =====
// Perform Instruction Cache Operation.

AArch64.IC(CacheOpScope opscope)
    regval = bits(64) UNKNOWN;
    AArch64.IC(regval, opscope);

// AArch64.IC()
// =====
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
    CacheRecord cache;
    AccType acctype = AccType_IC;

    cache.acctype = acctype;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        if opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR_EL2.FB == '1') then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = regval;
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        bits(64) vaddress = regval;
        need_translate = ICInstNeedsTranslation(opscope);

        cache.vaddress = regval;
        cache.shareability = Shareability_NSH;
        cache.translated = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        iswrite = FALSE;
        wasaligned = TRUE;
        size = 0;
        memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);

        if IsFault(memaddrdesc) then
            AArch64.Abort(regval, memaddrdesc.fault);

        cache.paddress = memaddrdesc.paddress;
        CACHE_OP(cache);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/sysop/SysOp

```
// SysOp()
// =====

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
  case op1:CRn:CRm:op2 of
    when '000 0111 1000 000' return Sys_AT;    // S1E1R
    when '100 0111 1000 000' return Sys_AT;    // S1E2R
    when '110 0111 1000 000' return Sys_AT;    // S1E3R
    when '000 0111 1000 001' return Sys_AT;    // S1E1W
    when '100 0111 1000 001' return Sys_AT;    // S1E2W
    when '110 0111 1000 001' return Sys_AT;    // S1E3W
    when '000 0111 1000 010' return Sys_AT;    // S1E0R
    when '000 0111 1000 011' return Sys_AT;    // S1E0W
    when '100 0111 1000 100' return Sys_AT;    // S12E1R
    when '100 0111 1000 101' return Sys_AT;    // S12E1W
    when '100 0111 1000 110' return Sys_AT;    // S12E0R
    when '100 0111 1000 111' return Sys_AT;    // S12E0W
    when '011 0111 0100 001' return Sys_DC;    // ZVA
    when '000 0111 0110 001' return Sys_DC;    // IVAC
    when '000 0111 0110 010' return Sys_DC;    // ISW
    when '011 0111 1010 001' return Sys_DC;    // CVAC
    when '000 0111 1010 010' return Sys_DC;    // CSW
    when '011 0111 1011 001' return Sys_DC;    // CVAU
    when '011 0111 1110 001' return Sys_DC;    // CIVAC
    when '000 0111 1110 010' return Sys_DC;    // CISW
    when '011 0111 1101 001' return Sys_DC;    // CVADP
    when '000 0111 0001 000' return Sys_IC;    // IALLUIS
    when '000 0111 0101 000' return Sys_IC;    // IALLU
    when '011 0111 0101 001' return Sys_IC;    // IVAU
    when '100 1000 0000 001' return Sys_TLBI;   // IPAS2E1IS
    when '100 1000 0000 101' return Sys_TLBI;   // IPAS2LE1IS
    when '000 1000 0011 000' return Sys_TLBI;   // VMALLE1IS
    when '100 1000 0011 000' return Sys_TLBI;   // ALLE2IS
    when '110 1000 0011 000' return Sys_TLBI;   // ALLE3IS
    when '000 1000 0011 001' return Sys_TLBI;   // VAE1IS
    when '100 1000 0011 001' return Sys_TLBI;   // VAE2IS
    when '110 1000 0011 001' return Sys_TLBI;   // VAE3IS
    when '000 1000 0011 010' return Sys_TLBI;   // ASIDE1IS
    when '000 1000 0011 011' return Sys_TLBI;   // VAAE1IS
    when '100 1000 0011 100' return Sys_TLBI;   // ALLE1IS
    when '000 1000 0011 101' return Sys_TLBI;   // VALE1IS
    when '100 1000 0011 101' return Sys_TLBI;   // VALE2IS
    when '110 1000 0011 101' return Sys_TLBI;   // VALE3IS
    when '100 1000 0011 110' return Sys_TLBI;   // VMALLS12E1IS
    when '000 1000 0011 111' return Sys_TLBI;   // VAALE1IS
    when '100 1000 0100 001' return Sys_TLBI;   // IPAS2E1
    when '100 1000 0100 101' return Sys_TLBI;   // IPAS2LE1
    when '000 1000 0111 000' return Sys_TLBI;   // VMALLE1
    when '100 1000 0111 000' return Sys_TLBI;   // ALLE2
    when '110 1000 0111 000' return Sys_TLBI;   // ALLE3
    when '000 1000 0111 001' return Sys_TLBI;   // VAE1
    when '100 1000 0111 001' return Sys_TLBI;   // VAE2
    when '110 1000 0111 001' return Sys_TLBI;   // VAE3
    when '000 1000 0111 010' return Sys_TLBI;   // ASIDE1
    when '000 1000 0111 011' return Sys_TLBI;   // VAAE1
    when '100 1000 0111 100' return Sys_TLBI;   // ALLE1
    when '000 1000 0111 101' return Sys_TLBI;   // VALE1
    when '100 1000 0111 101' return Sys_TLBI;   // VALE2
    when '110 1000 0111 101' return Sys_TLBI;   // VALE3
    when '100 1000 0111 110' return Sys_TLBI;   // VMALLS12E1
    when '000 1000 0111 111' return Sys_TLBI;   // VAALE1
  return Sys_SYS;
```

Library pseudocode for aarch64/instrs/system/sysops/sysop/SystemOp

```
enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_ALL

```
// AArch32.DTLBI_ALL()
// =====
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.

AArch32.DTLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_DALL;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.attr = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_ASID

```
// AArch32.DTLBI_ASID()
// =====
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
    TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_DASID;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = TLBILevel\_Any;
    r.attr = attr;
    r.asid = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_VA

```
// AArch32.DTLBI_VA()
// =====
// Invalidate by VA all stage 1 data TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.DTLBI_VA(Shareability security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_DVA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid =

    TLBIRecord r;
    r.op = TLBIOp_DVA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Zeros(8) : Rt<7:0>;
    r.address =
    r.address.address = Zeros(32) : Rt<31:12> : (20) : Rt<31:12> : Zeros(12); (12);

    TLBI(r);
    if shareability !=

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_ALL

```
// AArch32.ITLBI_ALL()
// =====
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.

AArch32.ITLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_IALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel\_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_ASID

```
// AArch32.ITLBI_ASID()
// =====
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
    TLBMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_IASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel\_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_VA

```
// AArch32.ITLBI_VA()
// =====
// Invalidate by VA all stage 1 instruction TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_VA(Shareability security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_IVA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid =

    TLBIRecord r;
    r.op = TLBIOp_IVA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Zeros(8) : Rt<7:0>;
    r.address =
    r.address.address = Zeros(32) : Rt<31:12> : (20) : Rt<31:12> : Zeros(12); (12);

    TLBI(r);
    if shareability !=

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_ALL

```
// AArch32.TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op          = TLBIOp\_ALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel\_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_ASID

```
// AArch32.TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
    TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_ASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel\_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```


Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_IPAS2

```
// AArch32.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2};
    assert security == SS\_NonSecure;

    TLBIRecord r;
    r.op = TLBIOp_IPAS2;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;

    r.address.address =

    TLBIRecord r;
    r.op = TLBIOp\_IPAS2;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.address = Zeros(24) : Rt<27:0> : (12) : Rt<27:0> : Zeros(12);
    r.ipaspace = r.address.paspace = PAS_NonSecure;

    TLBI(r);
    if shareability != PAS\_NonSecure;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VA

```
// AArch32.TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_VA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid =

    TLBIRecord r;
    r.op = TLBIOp\_VA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Zeros(8) : Rt<7:0>;
    r.address =
r.address.address = Zeros(32) : Rt<31:12> : (20) : Rt<31:12> : Zeros(12); (12);

    TLBI(r);
    if shareability !=

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VAA

```
// AArch32.TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;

    r.address.address =

    TLBIRecord r;
    r.op = TLBIOp\_VAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.address = Zeros(32) : Rt<31:12> : (20) : Rt<31:12> : Zeros(12); (12);

    TLBI(r);
    if shareability !=

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VMALL

```
// AArch32.TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_VMALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel\_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VMALLS12

```
// AArch32.TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                    Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op          = TLBIOp\_VMALLS12;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel\_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_ALL

```
// AArch64.TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op          = TLBIOp\_ALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel\_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_ASID

```
// AArch64.TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
    TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_ASID;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel\_Any;
    r.attr         = attr;
    r.asid         = Xt<63:48>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_IPAS2

```
// AArch64.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};};

    TLBIRecord r;
    r.op = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl = Xt<47:44>;

    r.address.address = Xt<39:0>;

    TLBIRecord r;
    r.op = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.address = ZeroExtend(Xt<39:0> : Zeros(12));
(12);

    case security of
        when SS\_NonSecure
            r.ipaspace = r.address.paspace = PAS_NonSecure;
        when PAS\_NonSecure;
        when SS\_Secure
            r.ipaspace = if Xt<63> == '1' then r.address.paspace = if Xt<63> == '1' then PAS_NonSecure else
PAS_Secure;

    TLBI(r);
    if shareability != PAS\_NonSecure else PAS\_Secure;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
    return;
```



```

// AArch64.TLBI_RIPAS2()
// =====
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};};

    TLBIRecord r;
    r.op = TLBIOp\_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl = Xt<47:44>;

    bits(2) tg = Xt<47:46>;
    integer scale =

    TLBIRecord r;
    r.op = TLBIOp\_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;

    bits(2) tg = Xt<47:46>;
    integer scale = UInt(Xt<45:44>);
    integer num = UInt(Xt<43:39>);
    integer baseaddr = SInt(Xt<36:0>);

    bits(64) start_address;
    boolean valid;

    (valid, r.tg, r.address, r.end_address) = (valid, r.tg, start_address, r.end_address) = TLBIRange

    if !valid then return;

    r.address.address = start_address<51:0>;

    case security of
        when SS\_NonSecure
            r.ipaspace = r.address.paspace = PAS_NonSecure;
        when PAS\_NonSecure;
        when SS\_Secure
            r.ipaspace = if Xt<63> == '1' then r.address.paspace = if Xt<63> == '1' then PAS_NonSecure else

    TLBI(r);
    if shareability != PAS\_NonSecure else PAS\_Secure;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
    return;

```


Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RVA

```
// AArch64.TLBI_RVA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
```

```
AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};;
```

```
TLBIRecord r;
r.op = TLBIOp_RVA;
r.from_aarch64 = TRUE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.asid = Xt<63:48>;
r.ttl = Xt<47:44>;
```

```
bits(64) start_address;
boolean valid;
```

```
(valid, r.tg, start_address, r.end_address) =
```

```
TLBIRecord r;
r.op = TLBIOp\_RVA;
r.from_aarch64 = TRUE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.asid = Xt<63:48>;
```

```
boolean valid;
```

```
(valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);
```

```
if !valid then return; if !valid then return;
```

```
r.address.address = start_address<51:0>;
if security ==
```

```
TLBISS\_Secure(r);
&& Xt<63> == '0' then
    r.address.paspace = PAS_Secure;
else
    r.address.paspace = PAS_NonSecure;
```

```
TLBI(r);
if shareability != Shareability\_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RVAA

```
// AArch64.TLBI_RVAA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RVAA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBIOp_RVAA;
r.from_aarch64 = TRUE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.ttl = Xt<47:44>;

bits(2) tg = Xt<47:46>;
integer scale =

TLBIRecord r;
r.op = TLBIOp\_RVAA;
r.from_aarch64 = TRUE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;

bits(2) tg = Xt<47:46>;
integer scale = UInt(Xt<45:44>;
integer num = UInt(Xt<43:39>;
integer baseaddr = SInt(Xt<36:0>;

bits(64) start_address;
boolean valid;

(valid, r.tg, r.address, r.end_address) = (valid, r.tg, start_address, r.end_address) = TLBIRange

if !valid then return; if !valid then return;

r.address.address = start_address<51:0>;

TLBI(r);
if shareability !=

TLBI(r);
if shareability != Shareability\_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VA

```
// AArch64.TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
               Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_VA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Xt<63:48>;
    r.ttl = Xt<47:44>;

    r.address.address = Xt<39:0> ;

    TLBIRecord r;
    r.op = TLBIOp\_VA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Xt<63:48>;
    r.address = ZeroExtend(Xt<43:0> : Zeros(12));(12);

    TLBI(r);
    if shareability !=

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VAA

```
// AArch64.TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl = Xt<47:44>;

    r.address.address = Xt<39:0>;

    TLBIRecord r;
    r.op = TLBIOp\_VAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.address = ZeroExtend(Xt<43:0> : Zeros(12));(12);

    TLBI(r);
    if shareability !=

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VMALL

```
// AArch64.TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_VMALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel\_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VMALLS12

```
// AArch64.TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                     Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op          = TLBIOp\_VMALLS12;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel\_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/ASID_NONE

```
constant bits(16) ASID_NONE = Zeros();
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/Broadcast

```
// Broadcast()
// =====
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated shareability
// domain.

Broadcast(Shareability shareability, TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/HasLargeAddress

```
// HasLargeAddress()
// =====
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

boolean HasLargeAddress(Regime regime)
    if !Have52BitIPAAAndPASpaceExt() then
        return FALSE;
    case regime of
        when Regime\_EL3
            return TCR_EL3<32> == '1';
        when Regime\_EL2
            return TCR_EL2<32> == '1';
        when Regime\_EL20
            return TCR_EL2<59> == '1';
        when Regime\_EL10
            return TCR_EL1<59> == '1';
        otherwise
            Unreachable();
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/SecurityStateAtEL

```
// SecurityStateAtEL()
// =====
// Returns the effective security state at the exception level based off current settings.

SecurityState SecurityStateAtEL(bits(2) EL)
    if !HaveEL(EL3) then
        if boolean IMPLEMENTATION_DEFINED "Secure-only implementation" then
            return SS\_Secure;
        else
            return SS\_NonSecure;
    elsif EL == EL3 then
        return SS\_Secure;
    else
        // For EL2 call only when EL2 is enabled in current security state
        assert(EL != EL2 || EL2Enabled());
        if !ELUsingAArch32(EL3) then
            return if SCR_EL3.NS == '1' then SS\_NonSecure else SS\_Secure;
        else
            return if SCR.NS == '1' then SS\_NonSecure else SS\_Secure;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI

```
// TLBI()
// =====
// Performs TLB maintenance of operation on TLB to invalidate the matching transition table entries.

TLBI(TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBILevel

```
enumeration TLBILevel {
    TLBILevel_Any,
    TLBILevel_Last
};
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIMemAttr

```
enumeration TLBIMemAttr {
    TLBI_AllAttr,
    TLBI_ExcludeXS
};
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIOp

```
enumeration TLBIOp {
    TLBIOp_DALL,           // AArch32 Data TLBI operations - deprecated
    TLBIOp_DASID,
    TLBIOp_DVA,
    TLBIOp_IALL,           // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
};
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIRange

```
// TLBIRange()
// =====
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIRange(Regime regime, bits(64) Xt)
    boolean valid = TRUE;
    bits(64) start = Zeros(64);
    bits(64) end   = Zeros(64);

    bits(2) tg      = Xt<47:46>;
    integer scale   = UInt(Xt<45:44>);
    integer num     = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start, end);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start<52:16> = Xt<36:0>;
                start<63:53> = Replicate(Xt<36>, 11);
            else
                start<48:12> = Xt<36:0>;
                start<63:49> = Replicate(Xt<36>, 15);
        when '10' // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start<52:16> = Xt<36:0>;
                start<63:53> = Replicate(Xt<36>, 11);
            else
                start<50:14> = Xt<36:0>;
                start<63:51> = Replicate(Xt<36>, 13);
        when '11' // 64KB
            tg_bits = 16;
            start<52:16> = Xt<36:0>;
            start<63:53> = Replicate(Xt<36>, 11);
        otherwise
            Unreachable();

    integer range = (num+1) << (5*scale + 1 + tg_bits);
    end   = start + range<63:0>;

    if end<52> != start<52> then
        // overflow, saturate it
        end = Replicate(start<52>, 64-52) : Ones(52);

    return (valid, tg, start, end);
```


Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIRecord

```
type TLBIRecord is (
  TLBIOp          op,
  boolean          from_aarch64, // originated as an AArch64 operation
  TLBIOp          op,
  boolean          from_aarch64, // originated as an AArch64 operation
  SecurityState    security,
  Regime           regime,
  bits(16)         vmid,
  bits(16)         asid,
  TLBILevel        level,
  TLBIMemAttr      attr,
  PAspace          ipaspace, // For operations that take IPA as input address
  bits(64)         address, // input address, for range operations, start address
  attr,
  FullAddress      address, // VA/IPA/BaseAddress
  bits(64)         end_address, // for range operations, end address
  bits(2)          tg, // for range operations, translation granule
  bits(2)          tg, // for range operations - translation granule
  bits(4)          ttl, // transition table walk level holding the leaf entry for the address k
)
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_ALL

```
// TLBI_ALL()
// =====

TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
  if UsingAArch32() then
    AArch32.TLBI_ALL(security, regime, shareability, attr);
  else
    AArch64.TLBI_ALL(security, regime, shareability, attr);
  return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_ASID

```
// TLBI_ASID()
// =====

TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
  TLBIMemAttr attr, bits(N) reg)
  if UsingAArch32() then
    assert N == 32;
    AArch32.TLBI_ASID(security, regime, vmid, shareability, attr, reg<31:0>);
  else
    assert N == 64;
    AArch64.TLBI_ASID(security, regime, vmid, shareability, attr, reg<63:0>);
  return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_IPAS2

```
// TLBI_IPAS2()
// =====

TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
  Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(N) reg)
  if UsingAArch32() then
    assert N == 32;
    AArch32.TLBI_IPAS2(security, regime, vmid, shareability, level, attr, reg<31:0>);
  else
    assert N == 64;
    AArch64.TLBI_IPAS2(security, regime, vmid, shareability, level, attr, reg<63:0>);
  return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_RIPAS2

```
// TLBI_RIPAS2()
// =====

TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
            Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert !UsingAArch32();

    AArch64.TLBI\_RIPAS2(security, regime, vmid, shareability, level, attr, Xt);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_RVA

```
// TLBI_RVA()
// =====

TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
         Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert !UsingAArch32();

    AArch64.TLBI\_RVA(security, regime, vmid, shareability, level, attr, Xt);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_RVAA

```
// TLBI_RVAA()
// =====

TLBI_RVAA(SecurityState security, Regime regime, bits(16) vmid,
          Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert !UsingAArch32();

    AArch64.TLBI\_RVAA(security, regime, vmid, shareability, level, attr, Xt);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_VA

```
// TLBI_VA()
// =====

TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
        Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(N) reg)
    if UsingAArch32() then
        assert N == 32;
        AArch32.TLBI\_VA(security, regime, vmid, shareability, level, attr, reg<31:0>);
    else
        assert N == 64;
        AArch64.TLBI\_VA(security, regime, vmid, shareability, level, attr, reg<63:0>);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_VAA

```
// TLBI_VAA()
// =====

TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
         Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(N) reg)
    if UsingAArch32() then
        assert N == 32;
        AArch32.TLBI\_VAA(security, regime, vmid, shareability, level, attr, reg<31:0>);
    else
        assert N == 64;
        AArch64.TLBI\_VAA(security, regime, vmid, shareability, level, attr, reg<63:0>);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_VMALL

```
// TLBI_VMALL()
// =====

TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
           Shareability shareability, TLBIMemAttr attr)
if UsingAArch32() then
    AArch32.TLBI\_VMALL(security, regime, vmid, shareability, attr);
else
    AArch64.TLBI\_VMALL(security, regime, vmid, shareability, attr);
return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_VMALLS12

```
// TLBI_VMALLS12()
// =====

TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
              Shareability shareability, TLBIMemAttr attr)
if UsingAArch32() then
    AArch32.TLBI\_VMALLS12(security, regime, vmid, shareability, attr);
else
    AArch64.TLBI\_VMALLS12(security, regime, vmid, shareability, attr);
return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/VMID

```
// VMID[]
// =====
// Effective VMID.

bits(16) VMID[]
if EL2Enabled() then
    if !ELUsingAArch32(EL2) then
        if Have16bitVMID() && VTCR_EL2.VS == '1' then
            return VTTBR_EL2.VMID;
        else
            return ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    else
        return ZeroExtend(VTTBR.VMID, 16);
elsif HaveEL(EL2) && HaveSecureEL2Ext() then
    return Zeros(16);
else
    return VMID\_NONE;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/VMID_NONE

```
constant bits(16) VMID_NONE = Zeros();
```

Library pseudocode for aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```
enumeration VBitOp {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

Library pseudocode for aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```
enumeration CompareOp {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                       CompareOp_LE, CompareOp_LT};
```

Library pseudocode for aarch64/instrs/vector/logical/immediateop/ImmediateOp

```
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                         ImmediateOp_ORR, ImmediateOp_BIC};
```

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/Reduce

```
// Reduce()
// =====

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    boolean altfp = HaveAltFP() && !UsingAArch32() && FPCR.AH == '1';
    return Reduce(op, input, esize, altfp);

// Reduce()
// =====
// Perform the operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behaviour.

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize, boolean altfp)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input<esize-1:0>;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize, altfp);
    lo = Reduce(op, input<half-1:0>, esize, altfp);

    case op of
        when ReduceOp_FMINNUM
            result = FPMINum(lo, hi, FPCR[]);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, FPCR[]);
        when ReduceOp_FMIN
            result = FPMIN(lo, hi, FPCR[], altfp);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, FPCR[], altfp);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR[]);
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/ReduceOp

```
enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
    ReduceOp_FMIN, ReduceOp_FMAX,
    ReduceOp_FADD, ReduceOp_ADD};
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, AccType acctype, integer size)
    assert !ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert (UsingAArch32\(\) && size IN {2,4}) || size == 4;

    match = FALSE;

    for i = 0 to UInt(ID_AA64DFR0_EL1.BRPs)
        match_i = AArch64.BreakpointMatch(i, vaddress, acctype, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint\(\) then
        reason = DebugHalt\_Breakpoint;
        Halt(reason);
    elseif match then
        acctype = AccType\_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return NoFault();
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = NoFault();

    d_side = (acctype != AccType\_IFETCH);
    if HaveNV2Ext\(\) && acctype == AccType\_NV2REGISTER then
        mask = '0';
        generate_exception = AArch64.GenerateDebugExceptionsFrom(EL2, IsSecure(), mask) && MDSCR_EL1.MDE
    else
        generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch64.CheckBreakpoint(vaddress, acctype, size);

    return fault;
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                     boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());

    if acctype IN {AccType_TTW, AccType_IC, AccType_AT, AccType_ATPAN} then
        return NoFault();
    if acctype == AccType_DC then
        if !iswrite then
            return NoFault();

    match = FALSE;
    match_on_read = FALSE;
    ispriv = AArch64.AccessUsesEL(acctype) != EL0;

    for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
        if AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;
            if DBGWCR_EL1[i].LSC<0> == '1' then
                match_on_read = TRUE;

    if match && acctype == AccType_ATOMICRW then
        iswrite = !match_on_read;

    if match && HaltOnBreakpointOrWatchpoint() then
        if acctype != AccType_NONFAULT && acctype != AccType_CNOTFIRST then
            reason = DebugHalt_Watchpoint;
            EDWAR = vaddress;
            Halt(reason);
        else
            // Fault will be reported and cancelled
            return AArch64.DebugFault(acctype, iswrite);
    elseif match then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return NoFault();
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.BlockBase

```
// AArch64.BlockBase()
// =====
// Extract the address embedded in a block descriptor pointing to the base of
// a memory block

bits(52) AArch64.BlockBase(bits(64) descriptor, bit ds, TGx tgx, integer level)
    bits(52) blockbase = Zeros();

    if tgx == TGx\_4KB && level == 2 then
        blockbase<47:21> = descriptor<47:21>;
    elsif tgx == TGx\_4KB && level == 1 then
        blockbase<47:30> = descriptor<47:30>;
    elsif tgx == TGx\_4KB && level == 0 then
        blockbase<47:39> = descriptor<47:39>;
    elsif tgx == TGx\_16KB && level == 2 then
        blockbase<47:25> = descriptor<47:25>;
    elsif tgx == TGx\_16KB && level == 1 then
        blockbase<47:36> = descriptor<47:36>;
    elsif tgx == TGx\_64KB && level == 2 then
        blockbase<47:29> = descriptor<47:29>;
    elsif tgx == TGx\_64KB && level == 1 then
        blockbase<47:42> = descriptor<47:42>;
    else
        Unreachable();

    if Have52BitPAExt() && tgx == TGx\_64KB then
        blockbase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        blockbase<51:48> = descriptor<9:8>;descriptor<49:48>;

    return blockbase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.IASize

```
// AArch64.IASize()
// =====
// Retrieve the number of bits containing the input address

integer AArch64.IASize(bits(6) txsz)
    return 64 - UInt(txsz);
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.NextTableBase

```
// AArch64.NextTableBase()
// =====
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors

bits(52) AArch64.NextTableBase(bits(64) descriptor, bit ds, TGx tgx)
    bits(52) tablebase = Zeros();

    case tgx of
        when TGx\_4KB tablebase<47:12> = descriptor<47:12>;
        when TGx\_16KB tablebase<47:14> = descriptor<47:14>;
        when TGx\_64KB tablebase<47:16> = descriptor<47:16>;

    if Have52BitPAExt() && tgx == TGx\_64KB then
        tablebase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        tablebase<51:48> = descriptor<9:8>;descriptor<49:48>;

    return tablebase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.PageBase

```
// AArch64.PageBase()
// =====
// Extract the address embedded in a page descriptor pointing to the base of
// a memory page

bits(52) AArch64.PageBase(bits(64) descriptor, bit ds, TGx tgx)
    bits(52) pagebase = Zeros\(\);

    case tgx of
        when TGx\_4KB   pagebase<47:12> = descriptor<47:12>;
        when TGx\_16KB  pagebase<47:14> = descriptor<47:14>;
        when TGx\_64KB  pagebase<47:16> = descriptor<47:16>;

    if Have52BitPAExt\(\) && tgx == TGx\_64KB then
        pagebase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        pagebase<51:48> = descriptor<9:8>;descriptor<49:48>;

    return pagebase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.PhysicalAddressSize

```
// AArch64.PhysicalAddressSize()
// =====
// Retrieve the number of bits bounding the physical address

integer AArch64.PhysicalAddressSize(bits(3) encoded_ps, TGx tgx)
    integer ps;

    case encoded_ps of
        when '000' ps = 32;
        when '001' ps = 36;
        when '010' ps = 40;
        when '011' ps = 42;
        when '100' ps = 44;
        when '101' ps = 48;
        when '110' ps = 52;
        otherwise
            ps = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

    if tgx != TGx\_64KB && !Have52BitIPAAndPASpaceExt\(\) then
        max_ps = Min(48, AArch64.PAMax\(\));
    else
        max_ps = AArch64.PAMax\(\);

    return Min(ps, max_ps);
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S1StartLevel

```
// AArch64.S1StartLevel()
// =====
// Compute the initial lookup level when performing a stage 1 translation
// table walk

integer AArch64.S1StartLevel(S1TTWParams walkparams)
    // Input Address size
    iasize = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride = granulebits - 3;

    return FINAL\_LEVEL - (((iasize-1) - granulebits) DIV stride);
```


Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S2SLTTEntryAddress

```
// AArch64.S2SLTTEntryAddress()
// =====
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S2SLTTEntryAddress(S2TTWParams walkparams, bits(52) ipa,
                                       FullAddress tablebase)

    startlevel = AArch64.S2StartLevel(walkparams);
    iasize     = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;
    levels      = FINAL\_LEVEL - startlevel;

    bits(52) index;
    lsb  = levels*stride + granulebits;
    msb  = iasize - 1;
    index = ZeroExtend(ipa<msb:lsb>:Zeros(3));

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S2StartLevel

```
// AArch64.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch64.S2StartLevel(S2TTWParams walkparams)
    case walkparams.tgx of
        when TGx\_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '000' return 2;
                when '001' return 1;
                when '010' return 0;
                when '011' return 3;
                when '100' return -1;
        when TGx\_16KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
                when '11' return 0;
        when TGx\_64KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.TTBaseAddress

```
// AArch64.TTBaseAddress()
// =====
// Retrieve the PA/IPA pointing to the base of the initial translation table

bits(52) AArch64.TTBaseAddress(bits(64) ttb, bits(6) txsz, bits(3) ps,
                                bit ds, TGx tgx, integer startlevel)
    bits(52) tablebase = Zeros();

    // Input Address size
    iasize      = AArch64.IASize(txsz);
    granulebits = TGxGranuleBits(tgx);
    stride      = granulebits - 3;
    levels      = FINAL\_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = iasize - (levels*stride + granulebits) + 3;

    if (Have52BitPAExt() && tgx == TGx\_64KB && ps == '110') || (ds == '1') then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttb<5:2>:ttb<47:6>;
    else
        tablebase<47:1> = ttb<47:1>;

    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.TTEntryAddress

```
// AArch64.TTEntryAddress()
// =====
// Compute translation table descriptor address within the table pointed to by
// the table base

FullAddress AArch64.TTEntryAddress(integer level, TGx tgx, bits(6) txsz,
                                     bits(64) ia, FullAddress tablebase)
    // Input Address size
    iasize      = AArch64.IASize(txsz);
    granulebits = TGxGranuleBits(tgx);
    stride      = granulebits - 3;
    levels      = FINAL\_LEVEL - level;

    bits(52) index;
    lsb      = levels*stride + granulebits;
    msb      = Min(iasize - 1, lsb + stride - 1);
    index = ZeroExtend(ia<msb:lsb>:Zeros(3));

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.AddrTop

```
// AArch64.AddrTop()
// =====
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

integer AArch64.AddrTop(bit tbid, AccType acctype, bit tbi)
    if tbid == '1' && acctype == AccType_IFETCH then
        return 63;

    if tbi == '1' then
        return 55;
    else
        return 63;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.ContiguousBitFaults

```
// AArch64.ContiguousBitFaults()
// =====
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

boolean AArch64.ContiguousBitFaults(bits(6) txsz, TGx tgx, integer level)
    // Input Address size
    iasize = AArch64.IASize(txsz);
    // Translation size
    tsize = TranslationSize(tgx, level) + AArch64.ContiguousSizeLog2(tgx, level);

    fault = boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit";

    return tsize > iasize && fault;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// =====
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)
    FaultRecord fault;

    fault.statuscode = Fault_Debug;
    fault.acctype = acctype;
    fault.write = iswrite;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    return fault;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.ExclusiveFault

```
// AArch64.ExclusiveFault()
// =====

FaultRecord AArch64.ExclusiveFault(AccType acctype, boolean iswrite,
                                   boolean secondstage, boolean s2fslwalk)
    FaultRecord fault;

    fault.statuscode = Fault_Exclusive;
    fault.acctype = acctype;
    fault.write = iswrite;
    fault.secondstage = secondstage;
    fault.s2fslwalk = s2fslwalk;

    return fault;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.IPAIsOutOfRange

```
// AArch64.IPAIsOutOfRange()
// =====
// Check bits not resolved by translation are ZERO

boolean AArch64.IPAIsOutOfRange(bits(52) ipa, S2TTWParams walkparams)
    //Input Address size
    iasize = AArch64.IASize(walkparams.txsz);

    if iasize < 52 then
        return !IsZero(ipa<51:iasize>);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.OAOutOfRange

```
// AArch64.OAOutOfRange()
// =====
// Returns whether output address is expressed in the configured size number of bits

boolean AArch64.OAOutOfRange(TTWState walkstate, bits(3) ps, TGx tgx)
    // Output Address size
    oasize = AArch64.PhysicalAddressSize(ps, tgx);
    baseaddress = walkstate.baseaddress.address;

    if oasize < 52 then
        return !IsZero(baseaddress<51:oasize>);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1HasAlignmentFault

```
// AArch64.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S1HasAlignmentFault(AccType acctype, boolean aligned,
                                     bit ntlsmid, MemoryAttributes memattrs)
    if acctype == AccType\_IFETCH || memattrs.memtype != MemType\_Device then
        return FALSE;

    if acctype == AccType\_A32LSMD && ntlsmid == '0' && memattrs.device != DeviceType\_GRE then
        return TRUE;

    return !aligned || acctype == AccType\_DCZVA;
```



```

// AArch64.S1HasPermissionsFault()
// =====
// Returns whether stage 1 access violates permissions of target memory

boolean AArch64.S1HasPermissionsFault(Regime regime, TTWState walkstate,
                                       S1TTWParams walkparams, boolean ispriv,
                                       AccType acctype, boolean iswrite)
permissions = walkstate.permissions;

if HasUnprivileged(regime) then
    // Apply leaf permissions
    case permissions.ap<2:1> of
        when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
        when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
        when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only

    // Apply hierarchical permissions
    case permissions.ap_table of
        when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
        when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
        when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
        when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access

    // Locations writable by unprivileged cannot be executed by privileged
    px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
    ux = NOT(permissions.uxn OR permissions.uxn_table);

    pan_access = !(acctype IN {AccType\_DC, AccType\_IFETCH, AccType\_AT, AccType\_NV2REGISTER});
    if HavePANExt() && pan_access && !(PSTATE.EL == EL1 && walkparams.nv1 == '1') then
        pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
else
    // Apply leaf permissions
    case permissions.ap<2> of
        when '0' (r,w) = ('1','1'); // No effect
        when '1' (r,w) = ('1','0'); // Read-only

    // Apply hierarchical permissions
    case permissions.ap_table<1> of
        when '0' (r,w) = ( r , w ); // No effect
        when '1' (r,w) = ( r ,'0'); // Read-only

    x = NOT(permissions.xn OR permissions.xn_table);

// Prevent execution from writable locations if WXN is set
x = x AND NOT(walkparams.wxn AND w);

// Prevent execution from Non-secure space by PE in secure state if SIF is set
if (AArch64.CurrentSecurityState() == SS\_Secure &&
    walkstate.baseaddress.paspace == PAS\_NonSecure) then
    x = x AND NOT(walkparams.sif);

if acctype == AccType\_IFETCH then
    if (ConstrainUnpredictable(Unpredictable\_INSTRDEVICE) == Constraint\_FAULT &&
        walkstate.memattrs.memtype == MemType\_Device) then
        return TRUE;

    return x == '0';
elseif acctype == AccType\_DC then
    if iswrite then
        return w == '0';
    else
        // DC from privileged context which do no write cannot permission fault
        return !ispriv && r == '0';
elseif acctype == AccType\_IC then
    // IC instructions do not write

```

```

    assert !iswrite;
    impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";

    // IC from privileged context cannot permission fault
    return !ispriv && r == '0' && impdef_ic_fault;
elseif iswrite then
    return w == '0';
else
    return r == '0';

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1InvalidTxSZ

```

// AArch64.S1InvalidTxSZ()
// =====
// Detect erroneous configuration of stage 1 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.S1InvalidTxSZ(S1TTWParams walkparams)
    mintxs = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    return UInt(walkparams.txsz) < mintxs || UInt(walkparams.txsz) > maxtxsz;

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2HasAlignmentFault

```

// AArch64.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S2HasAlignmentFault(AccType acctype, boolean aligned,
                                     MemoryAttributes memattrs)
    if acctype == AccType\_IFETCH || memattrs.memtype != MemType\_Device then
        return FALSE;

    return !aligned || acctype == AccType\_DCZVA;

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2HasPermissionsFault

```
// AArch64.S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory

boolean AArch64.S2HasPermissionsFault(boolean s2fslwalk, TTWState walkstate,
                                       S2TTWParams walkparams, boolean ispriv,
                                       AccType acctype, boolean iswrite)

    permissions = walkstate.permissions;
    memtype = walkstate.memattrs.memtype;

    r = permissions.s2ap<0>;
    w = permissions.s2ap<1>;

    case (permissions.s2xn:permissions.s2xnx) of
        when '00' (px,ux) = ('1','1');
        when '01' (px,ux) = ('0','1');
        when '10' (px,ux) = ('0','0');
        when '11' (px,ux) = ('1','0');

    x = if ispriv then px else ux;

    if s2fslwalk && walkparams.ptw == '1' && memtype == MemType_Device then
        return TRUE;
    elsif acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        return x == '0';
    elsif acctype == AccType_DC then
        // AArch32 DC maintenance instructions operating by VA cannot fault.
        if iswrite then
            return !UsingAArch32() && w == '0';
        else
            // DC from privileged context which do no write cannot permission fault
            return !UsingAArch32() && !ispriv && r == '0';
    elsif acctype == AccType_IC then
        // IC instructions do not write
        assert !iswrite;
        impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";

        // AArch32 IC maintenance instructions operating by VA cannot fault.
        // IC from privileged context cannot permission fault
        return !UsingAArch32() && !ispriv && r == '0' && impdef_ic_fault;
    elsif iswrite then
        return w == '0';
    else
        return r == '0';
```


Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2InconsistentSL

```
// AArch64.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

boolean AArch64.S2InconsistentSL(S2TTWParams walkparams)
    startlevel = AArch64.S2StartLevel(walkparams);
    levels     = FINAL\_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride     = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch64.IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2InvalidSL

```
// AArch64.S2InvalidSL()
// =====
// Detect invalid configuration of SL field

boolean AArch64.S2InvalidSL(S2TTWParams walkparams)
    case walkparams.tgx of
        when TGx\_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '1x1' return TRUE;
                when '11x' return TRUE;
                when '010' return AArch64.PAMax() < 44;
                when '011' return !HaveSmallTranslationTableExt();
                otherwise return FALSE;
        when TGx\_16KB
            case walkparams.ds:walkparams.sl0 of
                when '011' return TRUE;
                when '010' return AArch64.PAMax() < 42;
                otherwise return FALSE;
        when TGx\_64KB
            case walkparams.sl0 of
                when '11' return TRUE;
                when '10' return AArch64.PAMax() < 44;
                otherwise return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2InvalidTxSZ

```
// AArch64.S2InvalidTxSZ()
// =====
// Detect erroneous configuration of stage 2 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.S2InvalidTxSZ(S2TTWParams walkparams, boolean slaarch64)
    mintxs = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, slaarch64);
    maxtxs = AArch64.MaxTxSZ(walkparams.tgx);

    return UInt(walkparams.txsz) < mintxs || UInt(walkparams.txsz) > maxtxs;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.VAIsOutOfRange

```
// AArch64.VAIsOutOfRange()
// =====
// Check bits not resolved by translation are identical and of accepted value

boolean AArch64.VAIsOutOfRange(bits(64) va, AccType acctype, Regime regime,
                                S1TTWParams walkparams)
    addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);
    // Input Address size
    iasize = AArch64.IASize(walkparams.txsz);

    if HasUnprivileged(regime) then
        if AArch64.GetVARange(va) == VARange\_LOWER then
            return IsZero(va<addrtop:iasize>);
        else
            return IsOnes(va<addrtop:iasize>);
    else
        return IsZero(va<addrtop:iasize>);
```

Library pseudocode for aarch64/translation/vmsa_memattr/AArch64.IsS2ResultTagged

```
// AArch64.IsS2ResultTagged()
// =====
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

boolean AArch64.IsS2ResultTagged(MemoryAttributes s2_memattrs, boolean s1_tagged)
    return (
        s1_tagged &&
        (s2_memattrs.memtype == MemType\_Normal) &&
        (s2_memattrs.inner.attrs == MemAttr\_WB) &&
        (s2_memattrs.inner.hints == MemHint\_RWA) &&
        (!s2_memattrs.inner.transient) &&
        (s2_memattrs.outer.attrs == MemAttr\_WB) &&
        (s2_memattrs.outer.hints == MemHint\_RWA) &&
        (!s2_memattrs.outer.transient)
    );
```



```

// AArch64.S2ApplyFWBMemAttrs()
// =====
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_memattrs,
                                             bits(4) s2_attr, bits(2) s2_sh)

    MemoryAttributes memattrs;

    if s2_attr<2> == '0' then          // S2 Device, S1 any
        s2_device = DecodeDevice(s2_attr<1:0>);
        memattrs.memtype = MemType_Device;
        if s1_memattrs.memtype == memattrs.shareability = Shareability_OSH;
        if s1_memattrs.memtype == MemType_Device then
            memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_device);
        else
            memattrs.device = s2_device;

    elsif s2_attr<1:0> == '11' then    // S2 attr = S1 attr
        memattrs = s1_memattrs;

    elsif s2_attr<1:0> == '10' then    // Force writeback
        memattrs.memtype = s2_shareability = DecodeShareability(s2_sh);
        memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                         s2_shareability);
    elsif s2_attr<1:0> == '10' then    // Force writeback
        memattrs.memtype = MemType_Normal;
        memattrs.inner.attrs = MemAttr_WB;
        memattrs.outer.attrs = MemAttr_WB;

        if (s1_memattrs.memtype == MemType_Normal &&
            s1_memattrs.inner.attrs != MemAttr_NC) then
            memattrs.inner.hints = s1_memattrs.inner.hints;
            memattrs.inner.transient = s1_memattrs.inner.transient;
        else
            memattrs.inner.hints = MemHint_RWA;
            memattrs.inner.transient = FALSE;

        if (s1_memattrs.memtype == MemType_Normal &&
            s1_memattrs.outer.attrs != MemAttr_NC) then
            memattrs.outer.hints = s1_memattrs.outer.hints;
            memattrs.outer.transient = s1_memattrs.outer.transient;
        else
            memattrs.outer.hints = MemHint_RWA;
            memattrs.outer.transient = FALSE;

    else                                // Non-cacheable unless S1 is device
        if s1_memattrs.memtype == s2_shareability = DecodeShareability(s2_sh);
        memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                         s2_shareability);
    else                                // Non-cacheable unless S1 is device
        if s1_memattrs.memtype == MemType_Device then
            memattrs = s1_memattrs;
        else
            MemAttrHints cacheability_attr;
            cacheability_attr.attrs = MemAttr_NC;

            memattrs.memtype = MemType_Normal;
            memattrs.inner = cacheability_attr;
            memattrs.outer = cacheability_attr;

        s2_shareability = memattrs.shareability = DecodeShareabilityShareability_OSH(s2_sh);
        memattrs.shareability =;

        memattrs.tagged = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_shareability);
        memattrs.tagged = AArch64.IsS2ResultTagged(memattrs, s1_memattrs.tagged);

        memattrs.shareability = NormaliseShareability(memattrs);
(memattrs, s1_memattrs.tagged);

```

```
return memattrs;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.AccessUsesEL

```
// AArch64.AccessUsesEL()
// =====
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch64.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
    elsif acctype == AccType_NV2REGISTER then
        return EL2;
    else
        return PSTATE.EL;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.FaultAllowsSetAccessFlag

```
// AArch64.FaultAllowsSetAccessFlag()
// =====
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.FaultAllowsSetAccessFlag(FaultRecord fault)
    if fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode IN {Fault_Alignment, Fault_Permission} then
        return ConstrainUnpredictable(Unpredictable_AFUPDATE) == Constraint_TRUE;
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =====
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation

AddressDescriptor AArch64.FullTranslate(bits(64) va, AccType acctype,
                                         boolean iswrite, boolean aligned)

    fault = NoFault();
    fault.acctype = acctype;
    fault.write = iswrite;

    regime = TranslationRegime(PSTATE.EL, acctype);

    (fault, ipa) = AArch64.S1Translate(fault, regime, va, acctype, aligned, iswrite);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(va, fault);

    if regime == Regime_EL10 && EL2Enabled() then
        slaarch64 = TRUE;
        s2fslwalk = FALSE;
        (fault, pa) = AArch64.S2Translate(fault, ipa, slaarch64, s2fslwalk,
                                          acctype, aligned, iswrite);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(va, fault);
        else
            return pa;
    else
        return ipa;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.MemSwapTableDesc

```
// AArch64.MemSwapTableDesc()
// =====
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, bits(64)) (FaultRecord, bits(64)) AArch64.MemSwapTableDesc(FaultRecord fault,
bits(64) prev_desc, bits(64) new_desc, bit ee,
AddressDescriptor descupdateaddress)
descupdateaccess = AArch64.MemSwapTableDesc(FaultRecord fault,
bits(64) prev_desc, bits(64) new_desc, bit ee,
AddressDescriptor descupdateaddress)
descupdateaccess = CreateAccessDescriptor(AccType_ATOMICRW);

if ee == '1' then
    new_desc = BigEndianReverse(new_desc);
    prev_desc = BigEndianReverse(prev_desc);

// All observers in the shareability domain observe the
// following memory read and write accesses atomically.
(memstatus, mem_desc) = PhysMemRead(descupdateaddress, 8, descupdateaccess);
if IsFault(memstatus) then
    iswrite = FALSE;
    fault = HandleExternalTTWAbort(memstatus, iswrite, descupdateaddress,
descupdateaccess, 8, fault);
    if IsFault(fault.statuscode) then
        fault.acctype = AccType_ATOMICRW;
        return (fault, bits(64) UNKNOWN);
if mem_desc == prev_desc then
    memstatus = PhysMemWrite(descupdateaddress, 8,
descupdateaccess, new_desc);
    iswrite = TRUE;
    if IsFault(memstatus) then
        fault = HandleExternalTTWAbort(memstatus, iswrite, descupdateaddress,
descupdateaccess, 8, fault);
        fault.acctype = memstatus.acctype;
        if IsFault(fault.statuscode) then
            fault.acctype = AccType_ATOMICRW;
            return (fault, bits(64) UNKNOWN);
(prev_desc);

// All observers in the shareability domain observe the
// following memory read and write accesses atomically.
mem_desc = _Mem[descupdateaddress, 8, descupdateaccess, FALSE];
if mem_desc == prev_desc then
    _Mem[descupdateaddress, 8, descupdateaccess] = new_desc;
    mem_desc = new_desc;

if ee == '1' then
    mem_desc = BigEndianReverse(mem_desc);

assert mem_desc == new_desc;

return (fault, mem_desc);
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.S1DisabledOutput

```
// AArch64.S1DisabledOutput()
// =====
// Map the the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord fault,
    Regime regime, bits(64) va, AccType acctype, boolean aligned)

    walkparams = AArch64.GetS1TTWParams(regime, va);

    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    // Output Address
    FullAddress oa;
    oa.address = va<51:0>;
    case AArch64.CurrentSecurityState() of
        when SS\_Secure    oa.paspace = PAS\_Secure;
        when SS\_NonSecure oa.paspace = PAS\_NonSecure;

    MemoryAttributes memattrs;
    if regime == Regime\_EL10 && EL2Enabled() && walkparams.dc == '1' then
        MemAttrHints default_cacheability;
        default_cacheability.attrs = MemAttr\_WB;
        default_cacheability.hints = MemHint\_RWA;
        default_cacheability.transient = FALSE;

        memattrs.memtype = MemType\_Normal;
        memattrs.outer = default_cacheability;
        memattrs.inner = default_cacheability;
        memattrs.shareability = Shareability\_NSH;
        memattrs.tagged = walkparams.dct == '1';
        memattrs.xs = '0';
    elseif acctype == AccType\_IFETCH then
        MemAttrHints i_cache_attr;
        if AArch64.S1ICacheEnabled(regime) then
            i_cache_attr.attrs = MemAttr\_WT;
            i_cache_attr.hints = MemHint\_RA;
            i_cache_attr.transient = FALSE;
        else
            i_cache_attr.attrs = MemAttr\_NC;

        memattrs.memtype = MemType\_Normal;
        memattrs.outer = i_cache_attr;
        memattrs.inner = i_cache_attr;
        memattrs.shareability = Shareability\_OSH;
        memattrs.tagged = FALSE;
        memattrs.xs = '1';
    else
        memattrs.memtype = MemType\_Device;
        memattrs.device = DeviceType\_nGnRnE;
        memattrs.shareability = Shareability\_OSH;
        memattrs.xs = '1';

    fault.level = 0;
    addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);
    if !IsZero(va<addrtop:AArch64.PAMax()>) then
        fault.statuscode = Fault\_AddressSize;
    elseif AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmid, memattrs) then
        fault.statuscode = Fault\_Alignment;

    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN);
    else
        ipa = CreateAddressDescriptor(va, oa, memattrs);
        return (fault, ipa);
```



```

// AArch64.S1Translate()
// =====
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor)(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault,
Regime regime, bits(64) va, AccType acctype, boolean aligned,
boolean iswrite)

// Prepare fault fields in case a fault is detected
fault.secondstage = FALSE;
fault.s2fslwalk = FALSE;

if !AArch64.S1Enabled(regime) then
    return return AArch64.S1DisabledOutput(fault, regime, va, acctype, aligned);

walkparams = AArch64.S1DisabledOutput(fault, regime, va, acctype, aligned);

walkparams = AArch64.GetS1TTWParams(regime, va);

if (AArch64.VAIsOutOfRange(va, acctype, regime, walkparams) ||
    (AArch64.AccessUsesEL(acctype) == EL0 && walkparams.e0pd == '1')) then
    fault.statuscode = Fault_Translation;
    fault.level = 0;
    return (fault, return (fault, AddressDescriptor UNKNOWN);

(fault, descaddress, walkstate,
    descriptor) = AArch64.S1Walk(fault, walkparams, va, regime, acctype,
    iswrite);

if fault.statuscode != AddressDescriptor UNKNOWN);

(fault, descaddress, walkstate,
    descriptor) = AArch64.S1Walk(fault, walkparams, va, regime, acctype,
    iswrite);

if fault.statuscode != Fault_None then
    return (fault, return (fault, AddressDescriptor UNKNOWN);

ispriv = AddressDescriptor UNKNOWN);

ispriv = AArch64.AccessUsesEL(acctype) != EL0;

if acctype == AccType_IFETCH then
    // Flag the fetched instruction is from a guarded page
    SetInGuardedPage(walkstate.guardedpage == '1');

if AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsm,
    walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elseif IsAtomicRW(acctype) then
    if if AArch64.S1HasPermissionsFault(regime, walkstate, walkparams,
    ispriv, acctype, FALSE) then
        // The permission fault was not caused by lack of write permissions
        fault.statuscode = AArch64.S1HasPermissionsFault(regime, walkstate, walkparams,
    ispriv, acctype, FALSE) then
        // The permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = FALSE;
    elseif elseif AArch64.S1HasPermissionsFault(regime, walkstate, walkparams,
    ispriv, acctype, TRUE) then
        // The permission fault was caused by lack of write permissions
        fault.statuscode = AArch64.S1HasPermissionsFault(regime, walkstate, walkparams,
    ispriv, acctype, TRUE) then
        // The permission fault was caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = TRUE;
    elseif elseif AArch64.S1HasPermissionsFault(regime, walkstate, walkparams,
    ispriv, acctype, iswrite) then
        fault.statuscode = AArch64.S1HasPermissionsFault(regime, walkstate, walkparams,
    ispriv, acctype, iswrite) then

```

```

    fault.statuscode = Fault_Permission;

    new_desc = descriptor;
    if walkparams.ha == '1' && if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
        // Set descriptor AF bit
        new_desc<10> = '1';

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
    // memory do not reflect that and the access instigates a write.
    if (fault.statuscode == AArch64.FaultAllowsSetAccessFlag(fault) then
        // Set descriptor AF bit
        new_desc<10> = '1';

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
    // memory do not reflect that and the access instigates a write.
    if (fault.statuscode == Fault_None &&
        walkparams.ha == '1' &&
        walkparams.hd == '1' &&
        descriptor<51> == '1' && // Descriptor DBM bit
        (IsAtomicRW(acctype) || iswrite) &&
        !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
        // Clear descriptor AP[2] bit permitting stage 1 writes
        new_desc<7> = '0';

    // Either the access flag was clear or AP<2> is set
    if new_desc != descriptor then
        if regime == Regime_EL10 && EL2Enabled() then
            slaarch64 = TRUE;
            s2fslwalk = TRUE;
            aligned = TRUE;
            iswrite = TRUE;
            (s2fault, descupdateaddress) = (s2fault, descupdateaddress) = AArch64.S2Translate
            slaarch64, s2fslwalk, AArch64.S2Translate(fault, descaddress,
            slaarch64, s2fslwalk, AccType_ATOMICRW,
            aligned, iswrite);

            if s2fault.statuscode != Fault_None then
                return (s2fault, return (s2fault, AddressDescriptor UNKNOWN););
            else
                descupdateaddress = descaddress;

            (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                walkparams.ee, descupdateaddress);

            if fault.statuscode != AddressDescriptor UNKNOWN);
            else
                descupdateaddress = descaddress;

            (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                walkparams.ee, descupdateaddress);

            if fault.statuscode != Fault_None then
                return (fault, return (fault, AddressDescriptor UNKNOWN););

    // Output Address
    oa = Stage0A(walkstate.baseaddress, va, walkparams.tgx, walkstate.level);

    if (acctype == AddressDescriptor UNKNOWN);

    // Output Address
    oa = Stage0A(walkstate.baseaddress, va, walkparams.tgx, walkstate.level);

    if (acctype == AccType_IFETCH &&
        (walkstate.memattrs.memtype == MemType_Device || !AArch64.S1ICacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();

```

```

    memattrs.xs = walkstate.memattrs.xs;
elseif (acctype != AccType\_IFETCH && !AArch64.S1DCacheEnabled(regime) &&
        walkstate.memattrs.memtype == MemType\_Normal) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;

    // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
    // on the Tagged attribute
    if HaveMTE2Ext() && walkstate.memattrs.tagged then
        memattrs.tagged = ConstrainUnpredictableBool(Unpredictable\_S1CTAGGED);
else
    memattrs = walkstate.memattrs;

    // Shareability of target memory subject to stage 2 translation
    // is maintained as input to stage 2
    if regime == Regime\_EL10 && EL2Enabled() && HCR_EL2.VM == '1' then
        // Shareability of target memory subject to stage 2 translation
        // is maintained as input to stage 2
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = elseif (memattrs.memtype == NormaliseShareabilityMemType\_Device(memattr

    if acctype == ||
        (memattrs.inner.attrs == MemAttr\_NC && memattrs.outer.attrs == MemAttr\_NC)) then
        // If the target memory is Device memory or is Normal memory
        // inner Non-Cacheable outer Non-Cacheable then it is forced to Outer Shareable
        memattrs.shareability = Shareability\_OSH;

    if acctype == AccType\_ATOMICLS64 && memattrs.memtype == MemType\_Normal then
        if memattrs.inner.attrs != MemAttr\_NC || memattrs.outer.attrs != MemAttr\_NC then
            fault.statuscode = Fault\_Exclusive;
            return (fault, AddressDescriptor UNKNOWN);

    ipa = CreateAddressDescriptor(va, oa, memattrs);
;
    return (fault, AddressDescriptor UNKNOWN);

    ipa = CreateAddressDescriptor(va, oa, memattrs);
    return (fault, ipa);

```



```

// AArch64.S2Translate()
// =====
// Translate stage 1 IPA to PA and combine memory attributes

(FaultRecord, AddressDescriptor)(FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault,
    AddressDescriptor ipa, boolean slaarch64, boolean s2fslwalk, AArch64.S2Translate(acctype, boolean
    walkparams = AArch64.GetS2TTWParams(ipa.paddress.paspace, slaarch64);

    // Prepare fault fields in case a fault is detected
    fault.statuscode = FaultRecordFault_None fault,; // Ignore any faults from stage 1
    fault.secondstage = TRUE;
    fault.s2fslwalk = s2fslwalk;
    fault.ipaddress = ipa.paddress;

    if walkparams.vm != '1' then
        // Stage 2 translation is disabled
        fault.level = 0;
        if
            AddressDescriptorAArch64.S2HasAlignmentFault ipa, boolean slaarch64, boolean s2fslwalk, (acctype,
                fault.statuscode =
                AccTypeFault_Alignment acctype, boolean aligned, boolean iswrite)
        walkparams =;
        return (fault, AddressDescriptor UNKNOWN);
        else
            return (fault, ipa);

    if AArch64.GetS2TTWParams(ipa.paddress.paspace, slaarch64);

    // Prepare fault fields in case a fault is detected
    fault.statuscode = Fault_None; // Ignore any faults from stage 1
    fault.secondstage = TRUE;
    fault.s2fslwalk = s2fslwalk;
    fault.ipaddress = ipa.paddress;

    if walkparams.vm != '1' then
        // Stage 2 translation is disabled
        return (fault, ipa);

    if AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level = 0;
        return (fault, return (fault, AddressDescriptor UNKNOWN);

    (fault, descaddress, walkstate,
        descriptor) = AArch64.S2Walk(fault, ipa, walkparams, acctype, iswrite,
            slaarch64);

    if fault.statuscode != AddressDescriptor UNKNOWN);

    (fault, descaddress, walkstate,
        descriptor) = AArch64.S2Walk(fault, ipa, walkparams, acctype, iswrite,
            slaarch64);

    if fault.statuscode != Fault_None then
        return (fault, return (fault, AddressDescriptor UNKNOWN);

    ispriv = AddressDescriptor UNKNOWN);

    ispriv = AArch64.AccessUsesEL(acctype) != EL0;

    if AArch64.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif IsAtomicRW(acctype) then
        if if AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, walkparams,
            ispriv, acctype, FALSE) then
            // The permission fault was not caused by lack of write permissions
            fault.statuscode = AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, walkparams,
                ispriv, acctype, FALSE) then
            // The permission fault was not caused by lack of write permissions
            fault.statuscode = Fault_Permission;

```

```

        fault.write = FALSE;
    elsif AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, walkparams,
                                         ispriv, acctype, TRUE) then
        // The permission fault was caused by lack of write permissions.
        // However, HW updates, which are atomic writes for stage 1
        // descriptors, permissions fault reflect the original access.
        fault.statuscode = AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, walkparams,
                                                         ispriv, acctype, TRUE) then
        // The permission fault was caused by lack of write permissions.
        // However, HW updates, which are atomic writes for stage 1
        // descriptors, permissions fault reflect the original access.
        fault.statuscode = Fault_Permission;
        if !fault.s2fslwalk then
            fault.write = TRUE;
    elsif AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, walkparams,
                                         ispriv, acctype, iswrite) then
        fault.statuscode = AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, walkparams,
                                                         ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;

    new_desc = descriptor;
    if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
        // Set descriptor AF bit
        new_desc<10> = '1';

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
    // memory do not reflect that and the access instigates a write.
    if (fault.statuscode == AArch64.FaultAllowsSetAccessFlag(fault) then
        // Set descriptor AF bit
        new_desc<10> = '1';

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
    // memory do not reflect that and the access instigates a write.
    if (fault.statuscode == Fault_None &&
        walkparams.ha == '1' &&
        walkparams.hd == '1' &&
        descriptor<51> == '1' && // Descriptor DBM bit
        (IsAtomicRW(acctype) || iswrite) &&
        !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
        // Set descriptor S2AP[1] bit permitting stage 2 writes
        new_desc<7> = '1';

    // Either the access flag was clear or S2AP<1> is clear
    if new_desc != descriptor then
        (fault, mem_desc) = (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                                           walkparams.tee, descaddress);

    if fault.statuscode != AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                    walkparams.tee, descaddress);

    if fault.statuscode != Fault_None then
        return (fault, return (fault, AddressDescriptor UNKNOWN));

    ipa_64 = AddressDescriptor UNKNOWN);

    ipa_64 = ZeroExtend(ipa.paddress.address, 64);
    // Output Address
    oa = oa = Stage0A(walkstate.baseaddress, ipa_64, walkparams.tgx, walkstate.level);

    if ((s2fslwalk &&
        walkstate.memattrs.memtype == Stage0A(walkstate.baseaddress, ipa_64, walkparams.tgx, walkstate.level);

    if ((s2fslwalk &&
        walkstate.memattrs.memtype == MemType_Device && walkparams.ptw == '0') ||
        (acctype == AccType_IFETCH &&
        (walkstate.memattrs.memtype == MemType_Device || HCR_EL2.ID == '1')) ||

```

```

    (acctype != AccType\_IFETCH &&
     walkstate.memattrs.memtype == MemType\_Normal && HCR_EL2.CD == '1')) then
// Treat memory attributes as Normal Non-Cacheable
s2_memattrs = NormalNCMemAttr();
s2_memattrs.xs = walkstate.memattrs.xs;
else
    s2_memattrs = walkstate.memattrs;

if !s2fslwalk && acctype == AccType\_ATOMICS64 && s2_memattrs.memtype == MemType\_Normal then
    if s2_memattrs.inner.attrs != MemAttr\_NC || s2_memattrs.outer.attrs != MemAttr\_NC then
        fault.statuscode = Fault\_Exclusive;
        return (fault, return (fault, AddressDescriptor UNKNOWN););

if walkparams.fwb == '0' then
    s2_memattrs = AddressDescriptor UNKNOWN);

    if walkparams.fwb == '0' then
        memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
    else
        memattrs = s2_memattrs;

    pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
(ipa.memattrs, s2_memattrs);

pa = CreateAddressDescriptor(ipa.vaddress, oa, s2_memattrs);
    return (fault, pa);

```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.TranslateAddress

```

// AArch64.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccType acctype,
                                           boolean iswrite, boolean aligned,
                                           integer size)

    result = AArch64.FullTranslate(va, acctype, iswrite, aligned);

    if !IsFault(result) then
        result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(va);

return result;

```

Library pseudocode for aarch64/translation/vmsa_tentry/AArch64.BlockDescSupported

```

// AArch64.BlockDescSupported()
// =====
// Determine whether a block descriptor is valid for the given granule size
// and level

boolean AArch64.BlockDescSupported(bit ds, TGx tgx, integer level)
    case tgx of
        when TGx\_4KB return level == 2 || level == 1 || (level == 0 && ds == '1');
        when TGx\_16KB return level == 2 || (level == 1 && ds == '1');
        when TGx\_64KB return level == 2 || (level == 1 && AArch64.PAMax() == 52);

    return FALSE;

```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.BlocknTFaults

```
// AArch64.BlocknTFaults()
// =====
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault

boolean AArch64.BlocknTFaults(bits(64) descriptor)
    if !HaveBlockBBM() then
        return FALSE;

    bbm_level = AArch64.BlockBBMSupportLevel();
    nT_faults = boolean IMPLEMENTATION_DEFINED "BBM level 1 or 2 support nT bit causes Translation Fault"

    return bbm_level IN {1, 2} && descriptor<16> == '1' && nT_faults;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.ContiguousBit

```
// AArch64.ContiguousBit()
// =====
// Get the value of the contiguous bit
// REVISIT AARCH-16801

bit AArch64.ContiguousBit(TGx tgx, integer level, bits(64) descriptor)
    if tgx == TGx_64KB && level == 1 && !Have52BitVAExt() then
        return '0'; // RES0
    if tgx == TGx_16KB && level == 1 then
        return '0'; // RES0
    if tgx == TGx_4KB && level == 0 then
        return '0'; // RES0

    return descriptor<52>;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.ContiguousSizeLog2

```
// AArch64.ContiguousSizeLog2()
// =====
// Given the translation granule and level, determine the number of descriptors
// to the logarithm base 2 that describe a contiguous output space

integer AArch64.ContiguousSizeLog2(TGx tgx, integer level)
    case tgx of
        when TGx_4KB return 4;
        when TGx_16KB return if level == 2 then 5 else 7;
        when TGx_64KB return 5;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.DecodeDescriptorType

```
// AArch64.DecodeDescriptorType()
// =====
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(64) descriptor, bit ds,
                                             TGx tgx, integer level)
    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Page;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<1:0> == '01' then
        if AArch64.BlockDescSupported(ds, tgx, level) then
            return DescriptorType_Block;
        else
            return DescriptorType_Invalid;
    else
        return DescriptorType_Invalid;
```


Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S1ApplyOutputPerms

```
// AArch64.S1ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyOutputPerms(Permissions permissions, bits(64) descriptor,
                                         Regime regime, S1TTWParams walkparams)
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        permissions.ap<2:1> = descriptor<7>:'0';
        permissions.pxn     = descriptor<54>;

        return permissions;

    if HasUnprivileged(regime) then
        permissions.ap<2:1> = descriptor<7:6>;
        permissions.uxn     = descriptor<54>;
        permissions.pxn     = descriptor<53>;
    else
        permissions.ap<2:1> = descriptor<7>:'1';
        permissions.xn      = descriptor<54>;

    // Descriptors marked with DBM set have the effective value of AP[2] cleared.
    // This implies no permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
        permissions.ap<2> = '0';

    return permissions;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S1ApplyTablePerms

```
// AArch64.S1ApplyTablePerms()
// =====
// Apply hierarchical permissions encoded in stage 1 table descriptors

Permissions AArch64.S1ApplyTablePerms(Permissions permissions, bits(64) descriptor,
                                       Regime regime, S1TTWParams walkparams)

    if walkparams.hpd == '1' then
        permissions.ap_table = Zeros();
        if HasUnprivileged(regime) then
            permissions.uxn_table = Zeros();
            permissions.pxn_table = Zeros();
        else
            permissions.xn_table = Zeros();

        return permissions;

    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        ap_table = descriptor<62>:'0';
        pxn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;

        return permissions;

    if HasUnprivileged(regime) then
        ap_table = descriptor<62:61>;
        uxn_table = descriptor<60>;
        pxn_table = descriptor<59>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    else
        ap_table = descriptor<62>:'0';
        xn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;

    return permissions;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S2ApplyOutputPerms

```
// AArch64.S2ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 2 page/block descriptors

Permissions AArch64.S2ApplyOutputPerms(bits(64) descriptor, S2TTWParams walkparams)
    Permissions permissions;

    permissions.s2ap = descriptor<7:6>;
    permissions.s2xn = descriptor<54>;

    if HaveExtendedExecuteNeverExt() then
        permissions.s2xnx = descriptor<53>;
    else
        permissions.s2xnx = '0';

    // Descriptors marked with DBM set have the effective value of S2AP[1] set.
    // This implies no permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
        permissions.s2ap<1> = '1';

    return permissions;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1InitialTTWState

```
// AArch64.S1InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 1

TTWState AArch64.S1InitialTTWState(S1TTWParams walkparams, bits(64) va,
                                   Regime regime)

    TTWState    walkstate;
    FullAddress tablebase;

    startlevel = AArch64.S1StartLevel(walkparams);
    ttbr       = AArch64.S1TTBR(regime, va);
    case AArch64.CurrentSecurityState() of
        when SS\_Secure    tablebase.paspace = PAS\_Secure;
        when SS\_NonSecure tablebase.paspace = PAS\_NonSecure;

    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz,
                                             walkparams.ps, walkparams.ds,
                                             walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                         walkparams.orgn);

    return walkstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateLast

```
// AArch64.S1NextWalkStateLast()
// =====
// Decode stage 1 page or block descriptor as output to this stage of translation

TTWState AArch64.S1NextWalkStateLast(TTWState currentstate, Regime regime,
                                       S1TTWParams walkparams, bits(64) descriptor)
    TTWState    nextstate;
    FullAddress baseaddress;

    if currentstate.level == FINAL_LEVEL then
        baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds,
                                                walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds,
                                                walkparams.tgx, currentstate.level);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the block from NS bit
        baseaddress.paspace = if descriptor<5> == '0' then PAS_Secure else PAS_NonSecure;
    else
        baseaddress.paspace = PAS_NonSecure;

    nextstate.istable    = FALSE;
    nextstate.level      = currentstate.level;
    nextstate.baseaddress = baseaddress;

    attrindx = descriptor<4:2>;
    sh = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    attr = MAIRAttr(UInt(attrindx), walkparams.mair);
    slaarch64 = TRUE;

    nextstate.memattrs    = S1DecodeMemAttrs(attr, sh, slaarch64);
    nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions,
                                                         descriptor, regime, walkparams);
    nextstate.contiguous  = AArch64.ContiguousBit(walkparams.tgx, currentstate.level,
                                                  descriptor);
    nextstate.guardedpage = descriptor<50>;

    return nextstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateTable

```
// AArch64.S1NextWalkStateTable()
// =====
// Decode stage 1 table descriptor to transition to the next level

TTWState AArch64.S1NextWalkStateTable(TTWState currentstate, Regime regime,
                                       S1TTWParams walkparams, bits(64) descriptor)

    TTWState    nextstate;
    FullAddress tablebase;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.ds,
                                              walkparams.tgx);
    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the next table from NSTable bit
        tablebase.paspace = if descriptor<63> == '0' then PAS_Secure else PAS_NonSecure;
    else
        // Otherwise bit 63 is RES0 and there is no NSTable bit
        tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable      = TRUE;
    nextstate.level       = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs     = currentstate.memattrs;
    nextstate.permissions = AArch64.S1ApplyTablePerms(currentstate.permissions,
                                                       descriptor, regime,
                                                       walkparams);

    return nextstate;
```



```

// AArch64.S1Walk()
// =====
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

((FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S1Walk(
    FaultRecord fault, FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S1Walk(
    FaultRecord fault, S1TTWParams walkparams, bits(64) va, Regime regime,
    AccType acctype, boolean iswrite)
if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
    fault.statuscode = Fault_Translation;
    fault.level = 0;
    return (fault, return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN));

    if PSTATE.EL == AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN);

    if PSTATE.EL == ELO && walkparams.nfd == '1' then
        if acctype == AccType_NONFAULT then
            fault.statuscode = Fault_Translation;
            fault.level = 0;
            return (fault, return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(64) UNKNOWN));

        if AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(64) UNKNOWN);

        if AArch64.S1InvalidTxSZ(walkparams) then
            fault.statuscode = Fault_Translation;
            fault.level = 0;
            return (fault, return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(64) UNKNOWN));

        walkstate = AArch64.S1InitialTTWState(walkparams, va, regime);

        // Detect Address Size Fault by TTB
        if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
            fault.statuscode = AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(64) UNKNOWN);

        walkstate = AArch64.S1InitialTTWState(walkparams, va, regime);

        // Detect Address Size Fault by TTB
        if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
            fault.statuscode = Fault_AddressSize;
            fault.level = 0;
            return (fault, return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(64) UNKNOWN));

        bits(64) descriptor;
        repeat
            fault.level = walkstate.level;

            FullAddress descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx,
                walkparams.txsz, va,
                walkstate.baseaddress);

            if ! AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(64) UNKNOWN);

            bits(64) descriptor;
            repeat
                fault.level = walkstate.level;

                FullAddress descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx,
                    walkparams.txsz, va,
                    walkstate.baseaddress);

            if ! AArch64.S1DCacheEnabled(regime) then

```

```

        walkmemattrs = NormalNCMemAttr();
        walkmemattrs.xs = walkstate.memattrs.xs;
    else
        walkmemattrs = walkstate.memattrs;

    // Shareability of target memory subject to stage 2 translation
    // is maintained as input to stage 2.
    if regime ==                      // If the target memory for translation table walks is Normal memory
        // inner Non-Cacheable outer Non-Cacheable then it is forced to Outer Shareable
        // Note: this behaviour is overridden for addresses subject to stage 2.
        if (walkmemattrs.inner.attrs == Regime_EL10MemAttr_NC &&&
            walkmemattrs.outer.attrs == EL2EnabledMemAttr_NC() && HCR_EL2.VM == '1' then
            walkmemattrs.shareability = walkstate.memattrs.shareability;
        else
            walkmemattrs.shareability =) then
            walkmemattrs.shareability = NormaliseShareabilityShareability_OSH(walkmemattrs);
;

    walkaddress =                      walkaddress = CreateAddressDescriptor(va, descaddress, walkmemattrs);

    if regime == CreateAddressDescriptor(va, descaddress, walkmemattrs);

    if regime == Regime_EL10 && EL2Enabled() then
        // Shareability of target memory subject to stage 2 translation
        // is maintained as input to stage 2
        if HCR_EL2.VM == '1' then
            walkaddress.memattrs.shareability = walkstate.memattrs.shareability;

    slaarch64 = TRUE;
    s2fslwalk = TRUE;
    aligned = TRUE;
    iswrite = FALSE;
    (s2fault, s2walkaddress) =                      (s2fault, s2walkaddress) = AArch64.S2Translate(fault, v
        slaarch64, s2fslwalk, AArch64.S2Translate(fault, walkaddress,
        slaarch64, s2fslwalk, AccType_TTW,
        aligned, iswrite);

    if s2fault.statuscode != Fault_None then
        return (s2fault,                      return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKN
            bits(64) UNKNOWN);

    (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

    if fault.statuscode != AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN);

    (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

    if fault.statuscode != Fault_None then
        return (fault,                      return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(64) UNKNOWN);

    desctype = AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN);

    desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds,
        walkparams.tgx, walkstate.level);

    case desctype of
        when DescriptorType_Table
            walkstate =walkstate = AArch64.S1NextWalkStateTable(walkstate, regime,
                walkparams, descriptor);

        // Detect Address Size Fault by table descriptor
        if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
            fault.statuscode = AArch64.S1NextWalkStateTable(walkstate, regime,

```



```

        walkparams, descriptor);

    // Detect Address Size Fault by table descriptor
    if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
        fault.statuscode = Fault_AddressSize;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(64) UNKNOWN);

    when AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN);

    when DescriptorType_Page, DescriptorType_Block
        walkstate = walkstate = AArch64.SINextWalkStateLast(walkstate, regime,
            walkparams, descriptor);

    when AArch64.SINextWalkStateLast(walkstate, regime,
        walkparams, descriptor);

    when DescriptorType_Invalid
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(64) UNKNOWN);

    otherwise AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN);

    otherwise
        Unreachable();

until descctype IN {DescriptorType_Page, DescriptorType_Block};

if (walkstate.contiguous == '1' &&
    AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx,
        walkstate.level)) then
    fault.statuscode = Fault_Translation;
elseif descctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
    fault.statuscode = Fault_Translation;
// Detect Address Size Fault by final output
elseif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
    fault.statuscode = AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
    fault.statuscode = Fault_AddressSize;
// Check descriptor AF bit
elseif descriptor<10> == '0'
    && walkparams.ha == '0'
    && (!(acctype IN {
        elseif descriptor<10> == '0' && walkparams.ha == '0' then
        fault.statuscode = AccType_IC, AccType_DC}))
    || boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations") then
    fault.statuscode = Fault_AccessFlag;
if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN);
then
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN);
else
    return (fault, walkaddress, walkstate, descriptor);

```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2InitialTTWState

```
// AArch64.S2InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 2

TTWState AArch64.S2InitialTTWState(S2TTWParams walkparams)
    TTWState walkstate;
    FullAddress tablebase;

    ttbr = VTTBR_EL2;
    startlevel = AArch64.S2StartLevel(walkparams);
    tablebase.paspace = PAS_NonSecure;
    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz,
                                              walkparams.ps, walkparams.ds,
                                              walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level = startlevel;
    walkstate.istable = TRUE;
    walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                      walkparams.orgn);

    return walkstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateLast

```
// AArch64.S2NextWalkStateLast()
// =====
// Decode stage 2 page or block descriptor as output to this stage of translation

TTWState AArch64.S2NextWalkStateLast(TTWState currentstate, S2TTWParams walkparams,
                                      AddressDescriptor ipa, bits(64) descriptor)

    TTWState    nextstate;
    FullAddress baseaddress;

    if AArch64.CurrentSecurityState() == SS_Secure then
        baseaddress.paspace = AArch64.SS2OutputPASpace(walkparams,
                                                         ipa.paddress.paspace);
    else
        baseaddress.paspace = PAS_NonSecure;

    if currentstate.level == FINAL_LEVEL then
        baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds,
                                                walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds,
                                                walkparams.tgx, currentstate.level);

    nextstate.istable    = FALSE;
    nextstate.level      = currentstate.level;
    nextstate.baseaddress = baseaddress;
    nextstate.permissions = AArch64.S2ApplyOutputPerms(descriptor, walkparams);

    s2_attr = descriptor<5:2>;
    s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;
    if walkparams.fwb == '1' then
        nextstate.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, s2_attr, s2_sh);
        if s2_attr<1:0> == '10' then // Force writeback
            nextstate.memattrs.xs = '0';
        else
            nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
    else
        nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh);
        nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
    nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, currentstate.level,
                                                descriptor);

    return nextstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateTable

```
// AArch64.S2NextWalkStateTable()
// =====
// Decode stage 2 table descriptor to transition to the next level

TTWState AArch64.S2NextWalkStateTable(TTWState currentstate,
                                       S2TTWParams walkparams,
                                       bits(64) descriptor)

    TTWState    nextstate;
    FullAddress tablebase;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.ds,
                                                walkparams.tgx);
    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable    = TRUE;
    nextstate.level      = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;

    return nextstate;
```



```

// AArch64.S2Walk()
// =====
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

((FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S2Walk(
    FaultRecord fault, AddressDescriptor ipa, FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S2Walk(
    FaultRecord fault, AddressDescriptor ipa, S2TTWParams walkparams,
    AccType acctype, boolean iswrite, boolean slaarch64)
if (AArch64.S2InvalidTxSZ(walkparams, slaarch64) ||
    AArch64.S2InvalidSL(walkparams) ||
    AArch64.S2InconsistentSL(walkparams)) then
    fault.statuscode = Fault_Translation;
    fault.level = 0;
    return (fault, return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN));

if AddressDescriptor UNKNOWN, TTWState UNKNOWN,
    bits(64) UNKNOWN);

if AArch64.CurrentSecurityState() == SS_Secure then
    walkstate = walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
else
    walkstate = AArch64.S2InitialTTWState(walkparams);

// Detect Address Size Fault by TTB
if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
    fault.statuscode = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
else
    walkstate = AArch64.S2InitialTTWState(walkparams);

// Detect Address Size Fault by TTB
if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
    fault.statuscode = Fault_AddressSize;
    fault.level = 0;
    return (fault, return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN));

bits(64) descriptor;
repeat
    fault.level = walkstate.level;

FullAddress descaddress;
if walkstate.level == AddressDescriptor UNKNOWN, TTWState UNKNOWN,
    bits(64) UNKNOWN);

bits(64) descriptor;
repeat
    fault.level = walkstate.level;

FullAddress descaddress;
if walkstate.level == AArch64.S2StartLevel(walkparams) then
    // Initial lookup might index into concatenated tables
    descaddress = descaddress = AArch64.S2SLTTEnterAddress(walkparams, ipa.paddress.address,
        walkstate.baseaddress);
else
    ipa_64 = AArch64.S2SLTTEnterAddress(walkparams, ipa.paddress.address,
        walkstate.baseaddress);
else
    ipa_64 = ZeroExtend(ipa.paddress.address, 64);
    descaddress = descaddress = AArch64.TTEnterAddress(walkstate.level, walkparams.tgx,
        walkparams.txsz, ipa_64,
        walkstate.baseaddress);

if HCR_EL2.CD == '1' then
    walkmemattrs = AArch64.TTEnterAddress(walkstate.level, walkparams.tgx,
        walkparams.txsz, ipa_64,
        walkstate.baseaddress);

if HCR_EL2.CD == '1' then

```

```

        walkmemattrs = NormalNCMemAttr();
        walkmemattrs.xs = walkstate.memattrs.xs;
    else
        walkmemattrs = walkstate.memattrs;

    // VA parameter is for the Abort() call on the other side of Mem
    walkaddress = // If the target memory for translation table walks is Normal memory
    // inner Non-Cacheable outer Non-Cacheable then it is forced to Outer Shareable
    if (walkmemattrs.inner.attrs == CreateAddressDescriptorMemAttr_NC(ipa.vaddress, descaddress,

    walkaddress.memattrs.shareability = &&
        walkmemattrs.outer.attrs == NormaliseShareabilityMemAttr_NC(walkaddress.memattrs);
    (fault, descriptor) =) then
        walkmemattrs.shareability = FetchDescriptorShareability_OSH(walkparams.ee, walkaddress, 1
;

    // VA parameter is for the Abort() call on the other side of Mem
    walkaddress = CreateAddressDescriptor(ipa.vaddress, descaddress, walkmemattrs);

    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

    if fault.statuscode != Fault_None then
        return (fault, return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(64) UNKNOWN);

    desctype = AddressDescriptor UNKNOWN, TTWState UNKNOWN,
        bits(64) UNKNOWN);

    desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds,
        walkparams.tgx, walkstate.level);

    case desctype of
        when DescriptorType_Table
            walkstate = walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams,
                descriptor);

            // Detect Address Size Fault by table descriptor
            if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
                fault.statuscode = AArch64.S2NextWalkStateTable(walkstate, walkparams,
                    descriptor);

            // Detect Address Size Fault by table descriptor
            if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
                fault.statuscode = Fault_AddressSize;
                return (fault, return (fault, AddressDescriptor UNKNOWN, TTWState
                    bits(64) UNKNOWN);

        when AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(64) UNKNOWN);

        when DescriptorType_Page, DescriptorType_Block
            walkstate = walkstate = AArch64.S2NextWalkStateLast(walkstate, walkparams,
                ipa, descriptor);

        when AArch64.S2NextWalkStateLast(walkstate, walkparams,
            ipa, descriptor);

        when DescriptorType_Invalid
            fault.statuscode = Fault_Translation;
            return (fault, return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(64) UNKNOWN);

        otherwise AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(64) UNKNOWN);

        otherwise
            Unreachable();

    until desctype IN {DescriptorType_Page, DescriptorType_Block};

```

```

if (walkstate.contiguous == '1' &&
    AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx,
        walkstate.level)) then
    fault.statuscode = Fault_Translation;
elseif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
    fault.statuscode = Fault_Translation;
// Detect Address Size Fault by final output
elseif elseif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
    fault.statuscode = AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx) then
    fault.statuscode = Fault_AddressSize;
// Check descriptor AF bit
elseif descriptor<10> == '0'
    && walkparams.ha == '0'
    && (!(acctype IN { elseif descriptor<10> == '0' && walkparams.ha == '0' then
        fault.statuscode = AccType_IC, AccType_DC}))
    || boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations") then
    fault.statuscode = Fault_AccessFlag;
if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
then
return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
else
    return (fault, walkaddress, walkstate, descriptor);

```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.SS2InitialTTWState

```

// AArch64.SS2InitialTTWState()
// =====
// Set properties of first access to translation tables in Secure stage 2

TTWState AArch64.SS2InitialTTWState(S2TTWParams walkparams, PAspace ipaspace)
    TTWState walkstate;
    FullAddress tablebase;

    if ipaspace == PAS_Secure then
        ttbr = VSTTBR_EL2;
    else
        ttbr = VTTBR_EL2;

    if ipaspace == PAS_Secure then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
    else
        if walkparams.nsw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;

    startlevel = AArch64.S2StartLevel(walkparams);
    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz,
        walkparams.ps, walkparams.ds,
        walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level = startlevel;
    walkstate.istable = TRUE;
    walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn,
        walkparams.orgn);

    return walkstate;

```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.SS2OutputPASpace

```
// AArch64.SS2OutputPASpace()
// =====
// Assign PA Space to output of Secure stage 2 translation

PASpace AArch64.SS2OutputPASpace(S2TTWParams walkparams, PASpace ipaspace)
    if ipaspace == PAS_Secure then
        if walkparams.<sw,sa> == '00' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
    else
        if walkparams.<sw,sa,nsw,nsa> == '0000' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.BBMSupportLevel

```
// AArch64.BBMSupportLevel()
// =====
// Returns the level of FEAT_BBM supported

integer AArch64.BlockBBMSupportLevel()
    if !HaveBlockBBM() then
        return integer UNKNOWN;
    else
        return integer IMPLEMENTATION_DEFINED "Block BBM support level";
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.CurrentSecurityState

```
// AArch64.CurrentSecurityState()
// =====
// Return security state of current EL

SecurityState AArch64.CurrentSecurityState()
    return SecurityStateAtEL(PSTATE.EL);
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.DecodeTG0

```
// AArch64.DecodeTG0()
// =====
// Decode granule size configuration bits TG0

TGx AArch64.DecodeTG0(bits(2) tg0)
    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "Reserved TG0 encoding granule size";

    case tg0 of
        when '00'    return TGx_4KB;
        when '01'    return TGx_64KB;
        when '10'    return TGx_16KB;
```


Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.DecodeTG1

```
// AArch64.DecodeTG1()
// =====
// Decode granule size configuration bits TG1

TGx AArch64.DecodeTG1(bits(2) tgl)
  if tgl == '00' then
    tgl = bits(2) IMPLEMENTATION_DEFINED "Reserved TG1 encoding granule size";

  case tgl of
    when '10'   return TGx_4KB;
    when '11'   return TGx_64KB;
    when '01'   return TGx_16KB;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetS1TTWParams

```
// AArch64.GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective controlling
// system registers.

S1TTWParams AArch64.GetS1TTWParams(Regime regime, bits(64) va)
  S1TTWParams walkparams;

  varange = AArch64.GetVARange(va);

  case regime of
    when Regime_EL3   walkparams = AArch64.S1TTWParamsEL3();
    when Regime_EL2   walkparams = AArch64.S1TTWParamsEL2();
    when Regime_EL20  walkparams = AArch64.S1TTWParamsEL20(varange);
    when Regime_EL10  walkparams = AArch64.S1TTWParamsEL10(varange);

  maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
  mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
  if UInt(walkparams.txsz) > maxtxsz then
    if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
      walkparams.txsz = maxtxsz<5:0>;
  elsif !Have52BitVAExt() && UInt(walkparams.txsz) < mintxsz then
    if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
      walkparams.txsz = mintxsz<5:0>;

  return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetS2TTWParams

```
// AArch64.GetS2TTWParams()
// =====
// Gather walk parameters for stage 2 translation

S2TTWParams AArch64.GetS2TTWParams(PASpace ipaspace, boolean slaarch64)
    S2TTWParams walkparams;

    ss = AArch64.CurrentSecurityState();
    if ss == SS_NonSecure then
        walkparams = AArch64.NSS2TTWParams(slaarch64);
    elseif HaveSecureEL2Ext() && ss == SS_Secure then
        walkparams = AArch64.SS2TTWParams(ipaspace, slaarch64);
    else
        Unreachable();

    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, slaarch64);
    if UInt(walkparams.txsz) > maxtxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
            walkparams.txsz = maxtxsz<5:0>;
    elseif !Have52BitPAExt() && UInt(walkparams.txsz) < mintxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
            walkparams.txsz = mintxsz<5:0>;

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetVARange

```
// AArch64.GetVARange()
// =====
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.

VARange AArch64.GetVARange(bits(64) va)
    if va<55> == '0' then
        return VARange_LOWER;
    else
        return VARange_UPPER;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.MaxTxSZ

```
// AArch64.MaxTxSZ()
// =====
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

integer AArch64.MaxTxSZ(TGx tgx)
    if HaveSmallTranslationTableExt() && !UsingAArch32() then
        case tgx of
            when TGx_4KB    return 48;
            when TGx_16KB   return 48;
            when TGx_64KB   return 47;
    return 39;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.NSS2TTWParams

```
// AArch64.NSS2TTWParams()
// =====
// Gather walk parameters specific for Non-secure stage 2 translation

S2TTWParams AArch64.NSS2TTWParams(boolean slaarch64)
    S2TTWParams walkparams;

    walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.tgx = AArch64.DecodeTG0(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.T0SZ;
    walkparams.sl0 = VTCR_EL2.SL0;
    walkparams.ps = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGNO;
    walkparams.sh = VTCR_EL2.SH0;
    walkparams.ee = SCTLR_EL2.EE;

    walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
    walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.PAMax

```
// AArch64.PAMax()
// =====
// Returns the IMPLEMENTATION_DEFINED maximum number of bits capable of representing
// physical address for this processor

integer AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1DCacheEnabled

```
// AArch64.S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

boolean AArch64.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL3 return SCTLR_EL3.C == '1';
        when Regime_EL2 return SCTLR_EL2.C == '1';
        when Regime_EL20 return SCTLR_EL2.C == '1';
        when Regime_EL10 return SCTLR_EL1.C == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1EPD

```
// AArch64.S1EPD()
// =====
// Determine whether stage 1 translation table walk is allowed for the VA range

bit AArch64.S1EPD(Regime regime, bits(64) va)
    assert HasUnprivileged(regime);
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL20 return if varange == VARange_LOWER then TCR_EL2.EPD0 else TCR_EL2.EPD1;
        when Regime_EL10 return if varange == VARange_LOWER then TCR_EL1.EPD0 else TCR_EL1.EPD1;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1Enabled

```
// AArch64.S1Enabled()
// =====
// Determine if stage 1 for the acting translation regime is enabled

boolean AArch64.S1Enabled(Regime regime)
    case regime of
        when Regime_EL3 return SCTLR_EL3.M == '1';
        when Regime_EL2 return SCTLR_EL2.M == '1';
        when Regime_EL20 return SCTLR_EL2.M == '1';
        when Regime_EL10 return (!EL2Enabled() || HCR_EL2.<DC,TGE> == '00') && SCTLR_EL1.M == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1ICacheEnabled

```
// AArch64.S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

boolean AArch64.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL3 return SCTLR_EL3.I == '1';
        when Regime_EL2 return SCTLR_EL2.I == '1';
        when Regime_EL20 return SCTLR_EL2.I == '1';
        when Regime_EL10 return SCTLR_EL1.I == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1MinTxSZ

```
// AArch64.S1MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1

integer AArch64.S1MinTxSZ(bit ds, TGx tgx)
    if (Have52BitVAExt() && tgx == TGx_64KB) || ds == '1' then
        return 12;

    return 16;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTBR

```
// AArch64.S1TTBR()
// =====
// Identify stage 1 table base register for the acting translation regime

bits(64) AArch64.S1TTBR(Regime regime, bits(64) va)
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL3 return TTBR0_EL3;
        when Regime_EL2 return TTBR0_EL2;
        when Regime_EL20 return if varange == VARange_LOWER then TTBR0_EL2 else TTBR1_EL2;
        when Regime_EL10 return if varange == VARange_LOWER then TTBR0_EL1 else TTBR1_EL1;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL10

```
// AArch64.S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

S1TTWParams AArch64.S1TTWParamsEL10(VARange varange)
    S1TTWParams walkparams;

    if varange == VARange_LOWER then
        walkparams.tgx = AArch64.DecodeTG0(TCR_EL1.TG0);
        walkparams.txsz = TCR_EL1.T0SZ;
        walkparams.irgn = TCR_EL1.IRGN0;
        walkparams.orgn = TCR_EL1.ORGNO;
        walkparams.sh = TCR_EL1.SH0;
        walkparams.tbi = TCR_EL1.TBI0;

        walkparams.nfd = if HaveSVE() then TCR_EL1.NFD0 else '0';
        walkparams.tbid = if HavePACExt() then TCR_EL1.TBID0 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL1.E0PD0 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL1.HPD0 else '0';
    else
        walkparams.tgx = AArch64.DecodeTG1(TCR_EL1.TG1);
        walkparams.txsz = TCR_EL1.T1SZ;
        walkparams.irgn = TCR_EL1.IRGN1;
        walkparams.orgn = TCR_EL1.ORG1;
        walkparams.sh = TCR_EL1.SH1;
        walkparams.tbi = TCR_EL1.TBI1;

        walkparams.nfd = if HaveSVE() then TCR_EL1.NFD1 else '0';
        walkparams.tbid = if HavePACExt() then TCR_EL1.TBID1 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL1.E0PD1 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL1.HPD1 else '0';

    walkparams.mair = MAIR_EL1;
    walkparams.wxn = SCTLR_EL1.WXN;
    walkparams.ps = TCR_EL1.IPS;
    walkparams.ee = SCTLR_EL1.EE;
    walkparams.sif = SCR_EL3.SIF;

    if EL2Enabled() then
        walkparams.dc = HCR_EL2.DC;
        walkparams.dct = if HaveMTE2Ext() then HCR_EL2.DCT else '0';

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmid = SCTLR_EL1.nTlSMID;
    else
        walkparams.ntlsmid = '1';

    if EL2Enabled() then
        if HCR_EL2.<NV,NV1> == '01' then
            case Constraint_NVNV1_Unpredictable(Unpredictable_NVNV1) of
                when Constraint_NVNV1_00 walkparams.nv1 = '0';
                when Constraint_NVNV1_01 walkparams.nv1 = '1';
                when Constraint_NVNV1_11 walkparams.nv1 = '1';
            else
                walkparams.nv1 = HCR_EL2.NV1;
        else
            walkparams.nv1 = '0';

    walkparams.epan = if HavePAN3Ext() then SCTLR_EL1.EPAN else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL1.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL1.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = TCR_EL1.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL2

```
// AArch64.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch64.S1TTWParamsEL2()
    S1TTWParams walkparams;

    walkparams.tgx = AArch64.DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.ps = TCR_EL2.PS;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGNO;
    walkparams.sh = TCR_EL2.SH0;
    walkparams.tbi = TCR_EL2.TBI;
    walkparams.mair = MAIR_EL2;
    walkparams.wxn = SCTLR_EL2.WXN;
    walkparams.ee = SCTLR_EL2.EE;
    walkparams.sif = SCR_EL3.SIF;

    walkparams.tbid = if HavePACExt() then TCR_EL2.TBID else '0';
    walkparams.hpd = if AArch64.HaveHPDExt() then TCR_EL2.HPD else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL20

```
// AArch64.S1TTWParamsEL20()
// =====
// Gather stage 1 translation table walk parameters for EL2&0 regime

S1TTWParams AArch64.S1TTWParamsEL20(VARange varange)
    S1TTWParams walkparams;

    if varange == VARange_LOWER then
        walkparams.tgx = AArch64.DecodeTG0(TCR_EL2.TG0);
        walkparams.txsz = TCR_EL2.T0SZ;
        walkparams.irgn = TCR_EL2.IRGN0;
        walkparams.orgn = TCR_EL2.ORGNO;
        walkparams.sh = TCR_EL2.SH0;
        walkparams.tbi = TCR_EL2.TBI0;

        walkparams.nfd = if HaveSVE() then TCR_EL2.NFD0 else '0';
        walkparams.tbid = if HavePACExt() then TCR_EL2.TBID0 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL2.E0PD0 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD0 else '0';
    else
        walkparams.tgx = AArch64.DecodeTG1(TCR_EL2.TG1);
        walkparams.txsz = TCR_EL2.T1SZ;
        walkparams.irgn = TCR_EL2.IRGN1;
        walkparams.orgn = TCR_EL2.ORG1;
        walkparams.sh = TCR_EL2.SH1;
        walkparams.tbi = TCR_EL2.TBI1;

        walkparams.nfd = if HaveSVE() then TCR_EL2.NFD1 else '0';
        walkparams.tbid = if HavePACExt() then TCR_EL2.TBID1 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL2.E0PD1 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD1 else '0';

    walkparams.mair = MAIR_EL2;
    walkparams.wxn = SCTL_EL2.WXN;
    walkparams.ps = TCR_EL2.IPS;
    walkparams.ee = SCTL_EL2.EE;
    walkparams.sif = SCR_EL3.SIF;

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmid = SCTL_EL2.nTlSMID;
    else
        walkparams.ntlsmid = '1';

    walkparams.epan = if HavePAN3Ext() then SCTL_EL2.EPAN else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL3

```
// AArch64.S1TTWParamsEL3()
// =====
// Gather stage 1 translation table walk parameters for EL3 regime

S1TTWParams AArch64.S1TTWParamsEL3()
    S1TTWParams walkparams;

    walkparams.tgx = AArch64.DecodeTG0(TCR_EL3.TG0);
    walkparams.txsz = TCR_EL3.T0SZ;
    walkparams.ps = TCR_EL3.PS;
    walkparams.irgn = TCR_EL3.IRGN0;
    walkparams.orgn = TCR_EL3.ORGNO;
    walkparams.sh = TCR_EL3.SH0;
    walkparams.tbi = TCR_EL3.TBI;
    walkparams.mair = MAIR_EL3;
    walkparams.wxn = SCTLR_EL3.WXN;
    walkparams.ee = SCTLR_EL3.EE;
    walkparams.sif = SCR_EL3.SIF;

    walkparams.tbid = if HavePACExt() then TCR_EL3.TBID else '0';
    walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL3.HPD else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL3.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL3.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = TCR_EL3.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S2MinTxSZ

```
// AArch64.S2MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2

integer AArch64.S2MinTxSZ(bit ds, TGx tgx, boolean slaarch64)
    ips = AArch64.PAMax();

    if Have52BitPAExt() && tgx != TGx_64KB && ds == '0' then
        ips = Min(48, AArch64.PAMax());

    min_txsz = 64 - ips;
    if !slaarch64 then
        // EL1 is AArch32
        min_txsz = Min(min_txsz, 24);

    return min_txsz;
```


Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.SS2TTWParams

```
// AArch64.SS2TTWParams()
// =====
// Gather walk parameters specific for secure stage 2 translation

S2TTWParams AArch64.SS2TTWParams(PASpace ipaspace, boolean slaarch64)
    S2TTWParams walkparams;
    assert AArch64.CurrentSecurityState() == SS_Secure;

    if ipaspace == PAS_Secure then
        walkparams.tgx = AArch64.DecodeTG0(VSTCR_EL2.TG0);
        walkparams.txsz = VSTCR_EL2.T0SZ;
        walkparams.sl0 = VSTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
            walkparams.sl2 = VSTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    elseif ipaspace == PAS_NonSecure then
        walkparams.tgx = AArch64.DecodeTG0(VTCR_EL2.TG0);
        walkparams.txsz = VTCR_EL2.T0SZ;
        walkparams.sl0 = VTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
            walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    else
        Unreachable();

    walkparams.sw = VSTCR_EL2.SW;
    walkparams.nsw = VTCR_EL2.NSW;
    walkparams.sa = VSTCR_EL2.SA;
    walkparams.nsa = VTCR_EL2.NSA;
    walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.ps = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGNO;
    walkparams.sh = VTCR_EL2.SH0;
    walkparams.ee = SCTLR_EL2.EE;

    walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
    walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.VAMax

```
// AArch64.VAMax()
// =====
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// the virtual address for this processor

integer AArch64.VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
```

Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// =====

ClearStickyErrors()
    EDSCR.TXU = '0';           // Clear TX underrun flag
    EDSCR.RX0 = '0';           // Clear RX overrun flag

    if Halted() then           // in Debug state
        EDSCR.IT0 = '0';       // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO) then
        return;
    EDSCR.ERR = '0';           // Clear cumulative error flag

    return;
```

Library pseudocode for shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =====
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =====

bits(2) DebugTargetFrom(boolean secure)
    if HaveEL(EL2) && (!secure || (HaveSecureEL2Ext() && (!HaveEL(EL3) || SCR_EL3.EEL2 == '1'))) then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && !HaveAArch64HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// =====
// Returns the state of the OS Double Lock.
// FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
// TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if !HaveDoubleLock() then
        return FALSE;
    elsif ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

Library pseudocode for shared/debug/OSLockStatus/OSLockStatus

```
// OSLockStatus()
// =====
// Returns the state of the OS Lock.

boolean OSLockStatus()
    return (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK) == '1';
```

Library pseudocode for shared/debug/SoftwareLockStatus/Component

```
enumeration Component {
    Component_PMU,
    Component_Debug,
    Component_CTI
};
```

Library pseudocode for shared/debug/SoftwareLockStatus/GetAccessComponent

```
// Returns the accessed component.
Component GetAccessComponent();
```

Library pseudocode for shared/debug/SoftwareLockStatus/SoftwareLockStatus

```
// SoftwareLockStatus()
// =====
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
    Component component = GetAccessComponent();
    if !HaveSoftwareLock(component) then
        return FALSE;
    case component of
        when Component_Debug
            return EDLSR.SLK == '1';
        when Component_PMU
            return PMLSR.SLK == '1';
        when Component_CTI
            return CTILSR.SLK == '1';
    otherwise
        Unreachable();
```

Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        return AllowExternalDebugAccess(IsAccessSecure());
    else
        return AllowExternalDebugAccess(ExternalSecureInvasiveDebugEnabled());

// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(boolean allow_secure)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() || ExternalInvasiveDebugEnabled() then
        if allow_secure then
            return TRUE;
        elseif HaveEL(EL3) then
            if ELUsingAArch32(EL3) then
                return SDCR.EDAD == '0';
            else
                return MDCR_EL3.EDAD == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        return AllowExternalPMUAccess(IsAccessSecure());
    else
        return AllowExternalPMUAccess(ExternalSecureNoninvasiveDebugEnabled());

// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is allowed for the given
// Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(boolean allow_secure)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() || ExternalNoninvasiveDebugEnabled() then
        if allow_secure then
            return TRUE;
        elseif HaveEL(EL3) then
            if ELUsingAArch32(EL3) then
                return SDCR.EPMAD == '0';
            else
                return MDCR_EL3.EPMAD == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

Library pseudocode for shared/debug/authentication/Debug_authentication

```
signal DBGEN;  
signal NIDEN;  
signal SPIDEN;  
signal SPNIDEN;
```

Library pseudocode for shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()  
// =====  
// The definition of this function is IMPLEMENTATION DEFINED.  
// In the recommended interface, this function returns the state of the DBGEN signal.  
  
boolean ExternalInvasiveDebugEnabled()  
    return DBGEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()  
// =====  
// Returns TRUE if Trace and PC Sample-based Profiling are allowed  
  
boolean ExternalNoninvasiveDebugEnabled()  
    return (ExternalNoninvasiveDebugEnabled() &&  
            (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled() ||  
             (ELUsingAArch32(EL1) && PSTATE.EL == EL0 && SDER.SUNIDEN == '1')));
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()  
// =====  
// This function returns TRUE if the FEAT_Debugv8p4 is implemented, otherwise this  
// function is IMPLEMENTATION DEFINED.  
// In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN  
// OR NIDEN) signal.  
  
boolean ExternalNoninvasiveDebugEnabled()  
    return !HaveNoninvasiveDebugAuth() || ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()  
// =====  
// The definition of this function is IMPLEMENTATION DEFINED.  
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.  
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.  
  
boolean ExternalSecureInvasiveDebugEnabled()  
    if !HaveEL(EL3) && !IsSecure() then return FALSE;  
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =====
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    if HaveNoninvasiveDebugAuth() then
        return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
    else
        return ExternalSecureInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/authentication/IsAccessSecure

```
// Returns TRUE when an access is Secure
boolean IsAccessSecure();
```

Library pseudocode for shared/debug/authentication/IsCorePowered

```
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.
boolean IsCorePowered();
```

Library pseudocode for shared/debug/breakpoint/CheckValidStateMatch

```
// CheckValidStateMatch()
// =====
// Checks for an invalid state match that will generate Constrained Unpredictable behaviour, otherwise
// returns Constraint_NONE.

(Constraint, bits(2), bit, bits(2)) CheckValidStateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean isbreakpoint,
boolean reserved = FALSE;

// Match 'Usr/Sys/Svc' only valid for AArch32 breakpoints
if (!isbreakpoint || !HaveAArch32EL(EL1)) && HMC:PxC == '000' && SSC != '11' then
    reserved = TRUE;

// Both EL3 and EL2 are not implemented
if !HaveEL(EL3) && !HaveEL(EL2) && (HMC != '0' || SSC != '00') then
    reserved = TRUE;

// EL3 is not implemented
if !HaveEL(EL3) && SSC IN {'01','10'} && HMC:SSC:PxC != '10100' then
    reserved = TRUE;

// EL3 using AArch64 only
if (!HaveEL(EL3) || !HaveAArch64HighestELUsingAArch32()) && HMC:SSC:PxC == '11000' then
    reserved = TRUE;

// EL2 is not implemented
if !HaveEL(EL2) && HMC:SSC:PxC == '11100' then
    reserved = TRUE;

// Secure EL2 is not implemented
if !HaveSecureEL2Ext() && (HMC:SSC:PxC) IN {'01100','10100','x11x1'} then
    reserved = TRUE;

// Values that are not allocated in any architecture version
if (HMC:SSC:PxC) IN {'01110','100x0','10110','11x10'} then
    reserved = TRUE;

if reserved then
    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then
        return (c, bits(2) UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

return (Constraint_NONE, SSC, HMC, PxC);
```

Library pseudocode for shared/debug/breakpoint/NumBreakpointsImplementedGetNumBreakpoints

```
// NumBreakpointsImplemented()
// =====
// GetNumBreakpoints()
// =====
// Returns the number of breakpoints implemented. This is indicated to software by
// DBGDIDR.BRPs in AArch32 state, and ID_AA64DFR0_EL1.BRPs in AArch64 state.

integer NumBreakpointsImplemented()
GetNumBreakpoints()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";
```

Library pseudocode for shared/debug/

breakpoint/**NumContextAwareBreakpointsImplemented****GetNumContextAwareBreakpoints**

```
// NumContextAwareBreakpointsImplemented()
// =====
// GetNumContextAwareBreakpoints()
// =====
// Returns the number of context-aware breakpoints implemented. This is indicated to software by
// DBGDIDR.CTX_CMPs in AArch32 state, and ID_AA64DFR0_EL1.CTX_CMPs in AArch64 state.

integer NumContextAwareBreakpointsImplemented()
GetNumContextAwareBreakpoints()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";
```

Library pseudocode for shared/debug/

breakpoint/**NumWatchpointsImplemented****GetNumWatchpoints**

```
// NumWatchpointsImplemented()
// =====
// GetNumWatchpoints()
// =====
// Returns the number of watchpoints implemented. This is indicated to software by
// DBGDIDR.WRPs in AArch32 state, and ID_AA64DFR0_EL1.WRPs in AArch64 state.

integer NumWatchpointsImplemented()
GetNumWatchpoints()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";
```

Library pseudocode for shared/debug/cti/CTI_SetEventLevel

```
// Set a Cross Trigger multi-cycle input event trigger to the specified level.
CTI_SetEventLevel(CrossTriggerIn id, signal level);
```

Library pseudocode for shared/debug/cti/CTI_SignalEvent

```
// Signal a discrete event on a Cross Trigger input event trigger.
CTI_SignalEvent(CrossTriggerIn id);
```

Library pseudocode for shared/debug/cti/CrossTrigger

```
enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,           CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,     CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,     CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,       CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,           CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,     CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,     CrossTriggerIn_TraceExtOut3};
```


Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =====

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    if ELUsingAArch32\(EL1\) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                   (commtx && DBGDCCINT.TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                   (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID\_COMMIRQ, if commirq then HIGH else LOW);

    return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRRX_EL0

```
// DBGDTRRX_EL0[] (external write)
// =====
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return;

if EDSCR.ERR == '1' then return; // Error flag set: ignore write

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write

if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
    EDSCR.RX0 = '1'; EDSCR.ERR = '1'; // Overrun condition: ignore write
    return;

EDSCR.RXfull = '1';
DTRRX = value;

if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0'; // See comments in EDITR[] (external write)
    if !UsingAArch32() then
        ExecuteA64(0xD5330501<31:0>); // A64 "MRS X1,DBGDTRRX_EL0"
        ExecuteA64(0xB8004401<31:0>); // A64 "STR W1,[X0],#4"
        X[1] = bits(64) UNKNOWN;
    else
        ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MRS R1,DBGDTRRXint"
        ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "STR R1,[R0],#4"
        R[1] = bits(32) UNKNOWN;
    // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.RXfull = bit UNKNOWN;
        DBGDTRRX_EL0 = bits(64) UNKNOWN;
    else
        // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
        assert EDSCR.RXfull == '0';

    EDSCR.ITE = '1'; // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_EL0[] (external read)
// =====

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
return DTRRX;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRTX_EL0

```
// DBGDTRTX_EL0[] (external read)
// =====
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                                // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;                        // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then                    // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1'; EDSCR.ERR = '1';                        // Underrun condition: block side-effects
        return value;                                           // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                        // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);                        // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "LDR R1,[R0],#4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(64) UNKNOWN;
        else
            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>);                    // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';
            if !UsingAArch32() then
                X[1] = bits(64) UNKNOWN;
            else
                R[1] = bits(32) UNKNOWN;
            EDSCR.ITE = '1';                                        // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// =====

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write
DTRTX = value;
return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTR_EL0

```
// DBGDTR_EL0[] (write)
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
  // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
  // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
  assert N IN {32,64};
  if EDSCR.TXfull == '1' then
    value = bits(N) UNKNOWN;
  // On a 64-bit write, implement a half-duplex channel
  if N == 64 then DTRRX = value<63:32>;
  DTRTX = value<31:0>;          // 32-bit or 64-bit write
  EDSCR.TXfull = '1';
  return;

// DBGDTR_EL0[] (read)
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
  // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
  // For MRS <Xt>,DBGDTR_EL0     N=64, X[t]=result
  assert N IN {32,64};
  bits(N) result;
  if EDSCR.RXfull == '0' then
    result = bits(N) UNKNOWN;
  else
    // On a 64-bit read, implement a half-duplex channel
    // NOTE: the word order is reversed on reads with regards to writes
    if N == 64 then result<63:32> = DTRTX;
    result<31:0> = DTRRX;
  EDSCR.RXfull = '0';
  return result;
```

Library pseudocode for shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

Library pseudocode for shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// =====
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
  if EDPRSR<6:5,0> != '001' then                                // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return;

  if EDSCR.ERR == '1' then return;                               // Error flag set: ignore write

  // The Software lock is OPTIONAL.
  if memory_mapped && EDLSR.SLK == '1' then return;             // Software lock locked: ignore write

  if !Halted() then return;                                     // Non-debug state: ignore write

  if EDSCR.ITE == '0' || EDSCR.MA == '1' then
    EDSCR.ITO = '1'; EDSCR.ERR = '1';                           // Overrun condition: block write
    return;

  // ITE indicates whether the processor is ready to accept another instruction; the processor
  // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
  // is no indication that the pipeline is empty (all instructions have completed). In this
  // pseudocode, the assumption is that only one instruction can be executed at a time,
  // meaning ITE acts like "InstrCompl".
  EDSCR.ITE = '0';

  if !UsingAArch32() then
    ExecuteA64(value);
  else
    ExecuteT32(value<15:0> /*hw1*/, value<31:16> /*hw2*/);

  EDSCR.ITE = '1';

return;
```



```

// DCPSInstruction()
// =====
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

case target_el of
    when EL1
        if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
        elsif EL2Enabled() && HCR_EL2.TGE == '1' then UNDEFINED;
        else handle_el = EL1;

    when EL2
        if !HaveEL(EL2) then UNDEFINED;
        elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
        elsif !IsSecureEL2Enabled() && IsSecure() then UNDEFINED;
        else handle_el = EL2;

    when EL3
        if EDSCR.SDD == '1' || !HaveEL(EL3) then UNDEFINED;
        handle_el = EL3;
    otherwise
        Unreachable();

from_secure = IsSecure();
if ELUsingAArch32(handle_el) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    assert UsingAArch32(); // Cannot move from AArch64 to AArch32
    case handle_el of
        when EL1AArch32.WriteMode(M32_Svc);
            if HavePANExt() && SCTLR.SPAN == '0' then
                PSTATE.PAN = '1';
        when EL2 AArch32.WriteMode(M32_Hyp);
        when EL3AArch32.WriteMode(M32_Monitor);
            if HavePANExt() then
                if !from_secure then
                    PSTATE.PAN = '0';
                elsif SCTLR.SPAN == '0' then
                    PSTATE.PAN = '1';
    if handle_el == EL2 then
        ELR_hyp = bits(32) UNKNOWN; HSR = bits(32) UNKNOWN;
    else
        LR = bits(32) UNKNOWN;
        SPSR[] = bits(32) UNKNOWN;
        PSTATE.E = SCTLR[].EE;
        DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;

else // Targeting AArch64
    if UsingAArch32() then
        AArch64.MaybeZeroRegisterUppers();
        MaybeZeroSVEUppers(target_el);
        PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;
        if HavePANExt() && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
            (handle_el == EL2 && HCR_EL2.E2H == '1' &&
            HCR_EL2.TGE == '1' && SCTLR_EL2.SPAN == '0')) then
            PSTATE.PAN = '1';
        ELR[] = bits(64) UNKNOWN; SPSR[] = bits(64) UNKNOWN; ESR[] = bits(64) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = '0';
        if HaveMTEExt() then PSTATE.TCO = '1';

UpdateEDSCRFields(); // Update EDSCR PE state flags
sync_errors = HaveIESB() && SCTLR[].IESB == '1';
if HaveDoubleFaultExt() && !UsingAArch32() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
// SCTLR[].IESB might be ignored in Debug state.
if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
if sync_errors then

```

```

    SynchronizeErrors();
return;

```

Library pseudocode for shared/debug/halting/DRPSInstruction

```

// DRPSInstruction()
// =====
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLRL[0].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLRL[0].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    bits(64) spsr = SPSR[0];
    SetPSTATEFromPSR(spsr);

    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;
    else
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;

    UpdateEDSCRFields(); // Update EDSCR PE state flags

return;

```

Library pseudocode for shared/debug/halting/DebugHalt

```

constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRRQ        = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';

```

Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```

DisableITRAndResumeInstructionPrefetch();

```

Library pseudocode for shared/debug/halting/ExecuteA64

```

// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);

```

Library pseudocode for shared/debug/halting/ExecuteT32

```

// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);

```


Library pseudocode for shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// =====

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001'; // Signal restarting
    EDESR<2:0> = '000'; // Clear any pending Halting debug events

    bits(64) new_pc;
    bits(64) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR);
        spsr = ZeroExtend(DSPSR);
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    if UsingAArch32() then
        SetPSTATEFromPSR(spsr<31:0>); // Can update privileged bits, even at EL0
    else
        SetPSTATEFromPSR(spsr); // Can update privileged bits, even at EL0

    boolean branch_conditional = FALSE;
    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
        // AArch32 branch
        BranchTo(new_pc<31:0>, BranchType_DBGEXIT, branch_conditional);
    else
        // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
        if spsr<4> == '1' && ConstrainUnpredictableBool(Unpredictable_RESTARTZEROUPPERPC) then
            new_pc<63:32> = Zeros();
        // A type of branch that is never predicted
        BranchTo(new_pc, BranchType_DBGEXIT, branch_conditional);

    (EDSCR.STATUS, EDPRSR.SDR) = ('000010', '1'); // Atomically signal restarted
    UpdateEDSCRFields(); // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
```

Library pseudocode for shared/debug/halting/Halt

```
// Halt()
// =====

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt); // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr();
    bits(32) spsr_32;
    bits(64) spsr_64;
    if UsingAArch32() then
        spsr_32 = GetPSRFromPSTATE(DebugState);
    else
        spsr_64 = GetPSRFromPSTATE(DebugState);

    if (HaveBTIExt() &&
        !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive, DebugHalt_Step_NoSyndrome,
                    DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &&
        ConstrainUnpredictableBool(Unpredictable_ZEROBTPE)) then
        if UsingAArch32() then
            spsr_32<11:10> = '00';
        else
            spsr_64<11:10> = '00';

    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr_32;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr_64;

    EDSCR.ITE = '1';
    EDSCR.IT0 = '0';
    if IsSecure() then
        EDSCR.SDD = '0'; // If entered in Secure state, allow debug
    elseif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        assert EDSCR.SDD == '1'; // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';

    // In Debug state:
    // * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
    //   are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.{IT,T} are ignored.
    // * PSTATE.IL is ignored and behave-as-if 0.
    // * PSTATE.BTYPE is ignored and behave-as-if 0.
    // * PSTATE.TCO is set 1.
    // * PSTATE.{UA0,PAN} are observable and not changed on entry into Debug state.
    if UsingAArch32() then
        PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
    else
        PSTATE.<SS,SSBS,D,A,I,F> = bits(6) UNKNOWN;
        PSTATE.TCO = '1';
        PSTATE.BTYPE = '00';
    PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason; // Signal entered Debug state
    UpdateEDSCRFields(); // Update EDSCR PE state flags.

return;
```

Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// =====
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

Library pseudocode for shared/debug/halting/Halted

```
// Halted()
// =====

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'}); // Halted
```

Library pseudocode for shared/debug/halting/HaltingAllowed

```
// HaltingAllowed()
// =====
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elseif IsSecure() then
        return ExternalSecureInvasiveDebugEnabled();
    else
        return ExternalInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/halting/Restarting

```
// Restarting()
// =====

boolean Restarting()
    return EDSCR.STATUS == '000001'; // Restarting
```

Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

```
StopInstructionPrefetchAndEnableITR();
```

Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// =====
// Update EDSCR PE state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;
        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        EDSCR.NS = if IsSecure() then '0' else '1';

        bits(4) RW;
        RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW<0> = RW<1>;
        else
            RW<0> = if UsingAArch32() then '0' else '1';
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0' && !IsSecureEL2Enabled()) then
            RW<2> = RW<1>;
        else
            RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
        if !HaveEL(EL3) then
            RW<3> = RW<2>;
        else
            RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;
    return;
```

Library pseudocode for shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that IsSecure() and PSTATE.EL are correct
    // for the exception target. When FEAT_Debugv8p2 is not implemented, this function might also be called
    // at any time.
    // for the exception target.
    base = if IsSecure() then 0 else 4;
    if HaltingAllowed() then
        if HaveExtendedECDebugEvents() then
            exception_exit = !exception_entry;
            ctrl = EDECCR<UInt>(PSTATE.EL) + base + 8>:EDECCR<UInt>(PSTATE.EL) + base>;
            case ctrl of
                when '00' halt = FALSE;
                when '01' halt = TRUE;
                when '10' halt = (exception_exit == TRUE);
                when '11' halt = (exception_entry == TRUE);
        else
            halt = (EDECCR<UInt>(PSTATE.EL) + base> == '1');
    if halt then Halt(DebugHalt_ExceptionCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// =====
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elseif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
```

Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()

    if (HaveDoPD() && CTIDEVCTL.OSUCE == '1')
    || (!HaveDoPD() && EDECR.OSUCE == '1')
    then
        if !Halted() then EDESR.OSUC = '1';
```

Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// =====
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// =====
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =====
// Called after reset

CheckResetCatch()
    if (HaveDoPD() && CTIDEVCTL.RCE == '1') || (!HaveDoPD() && EDECR.RCE == '1') then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32\(EL1\) then DBG0SLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed\(\) && EDCR.TDA == '1' && os_lock == '0' then
        Halt\(DebugHalt\_SoftwareAccess\);
```

Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// =====

ExternalDebugRequest()
    if HaltingAllowed\(\) then
        Halt\(DebugHalt\_EDBGRQ\);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();
```

Library pseudocode for shared/debug/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// =====

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    //
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exception.
    //
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted\(\); // Cannot come out of reset halted
    active = EDCR.SS == '1' && !Halted\(\);

    if active && reset then // Coming out of reset with EDCR.SS set
        EDCR.SS = '1';
    elsif active && HaltingAllowed\(\) then
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled\(\);
        else
            advance = TRUE;
        if advance then EDCR.SS = '1';

    return;
```

Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =====
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    if Havev8p4Debug() then
        if target == EL3 || IsSecure() then
            int_dis = (EDSCR.INTdis[0] == '1' && ExternalSecureInvasiveDebugEnabled());
        else
            int_dis = (EDSCR.INTdis[0] == '1');
    else
        case target of
            when EL3
                int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
            when EL2
                int_dis = (EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled());
            when EL1
                if IsSecure() then
                    int_dis = (EDSCR.INTdis == '1x' && ExternalSecureInvasiveDebugEnabled());
                else
                    int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
        return int_dis;
```

Library pseudocode for shared/debug/interrupts/InterruptID

```
enumeration InterruptID {InterruptID_PMUIRQ, InterruptID_COMMIHQ, InterruptID_CTIHQ,
                          InterruptID_COMMRX, InterruptID_COMMTX};
```

Library pseudocode for shared/debug/interrupts/SetInterruptRequestLevel

```
// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);
```

Library pseudocode for shared/debug/ pmu/NumEventCountersImplementedGetNumEventCounters

```
// NumEventCountersImplemented()
// =====
// GetNumEventCounters()
// =====
// Returns the number of event counters implemented. This is indicated to software at the
// highest Exception level by PMCR.N in AArch32 state, and PMCR_EL0.N in AArch64 state.

integer NumEventCountersImplemented()
GetNumEventCounters()
    return integer IMPLEMENTATION_DEFINED "Number of event counters";
```

Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// =====

CreatePCSample()
// In a simple sequential execution of the program, CreatePCSample is executed each time the PE
// executes an instruction that can be sampled. An implementation is not constrained such that
// reads of EDPCSRlo return the current values of PC, etc.

pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
pc_sample.pc = ThisInstrAddr();
pc_sample.el = PSTATE.EL;
pc_sample.rw = if UsingAArch32() then '0' else '1';
pc_sample.ns = if IsSecure() then '0' else '1';
pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1<31:0>;
pc_sample.has_el2 = EL2Enabled();

if EL2Enabled() then
    if ELUsingAArch32(EL2) then
        pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
    elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
        pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    else
        pc_sample.vmid = VTTBR_EL2.VMID;
    if (HaveVirtHostExt() || HaveV82Debug()) && !ELUsingAArch32(EL2) then
        pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
    else
        pc_sample.contextidr_el2 = bits(32) UNKNOWN;
    pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
return;
```


Library pseudocode for shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// =====

bits(32) EDPCSRlo[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0'; // Software locked: no side-effects

if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        if HaveVirtHostExt() && EDSCR.SC2 == '1' then
            EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            EDPCSRhi.EL = pc_sample.el;
            EDPCSRhi.NS = pc_sample.ns;
        else
            EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if (HaveVirtHostExt() || HaveV82Debug()) && EDSCR.SC2 == '1' then
                EDVIDSR = (if HaveEL(EL2) && pc_sample.ns == '1' then pc_sample.contextidr_el2
                    else bits(32) UNKNOWN);
            else
                if HaveEL(EL2) && pc_sample.ns == '1' && pc_sample.el IN {EL1,EL0} then
                    EDVIDSR.VMID = pc_sample.vmid;
                else
                    EDVIDSR.VMID = Zeros();
                    EDVIDSR.NS = pc_sample.ns;
                    EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
                    EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
                    // The conditions for setting HV are not specified if PCSRhi is zero.
                    // An example implementation may be "pc_sample.rw".
                    EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
        else
            sample = Ones(32);
            if update then
                EDPCSRhi = bits(32) UNKNOWN;
                EDCIDSR = bits(32) UNKNOWN;
                EDVIDSR = bits(32) UNKNOWN;

    return sample;
```

Library pseudocode for shared/debug/samplebasedprofiling/PCSample

```
type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    bit ns,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)

PCSample pc_sample;
```

Library pseudocode for shared/debug/samplebasedprofiling/PMPCSR

```
// PMPCSR[] (read)
// =====

bits(32) PMPCSR[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || PMLSR.SLK == '0'; // Software locked: no side-effects

    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            PMPCSR.EL = pc_sample.el;
            PMPCSR.NS = pc_sample.ns;

            PMCID1SR = pc_sample.contextidr;
            PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;

            PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h
                            then pc_sample.vmid else bits(16) UNKNOWN);
        else
            sample = Ones(32);
            if update then
                PMPCSR<55:32> = bits(24) UNKNOWN;
                PMPCSR.EL = bits(2) UNKNOWN;
                PMPCSR.NS = bit UNKNOWN;

                PMCID1SR = bits(32) UNKNOWN;
                PMCID2SR = bits(32) UNKNOWN;

                PMVIDSR.VMID = bits(16) UNKNOWN;

    return sample;
```

Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// =====
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    step_enabled = !ELUsingAArch32\(DebugTarget\(\)\) && AArch64.GenerateDebugExceptions\(\) && MDSCR_EL1.SS ==
    if step_enabled && PSTATE.SS == '0' then
        AArch64.SoftwareStepException\(\);
```

Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// =====
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    if UsingAArch32() then
        assert N == 32;
    else
        assert N == 64;

    assert Halted() || Restarting() || PSTATE.EL != EL0;

    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();

    if IllegalExceptionReturn(spsr) then
        dest = PSTATE.EL;
    else
        (valid, dest) = ELFromSPSR(spsr); assert valid;

    dest_is_secure = IsSecureBelowEL3() || dest == EL3;
    dest_using_32 = (if dest == EL0 then spsr<4> == '1' else ELUsingAArch32(dest));
    if dest_using_32 then
        enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, dest_is_secure);
    else
        mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, dest_is_secure, mask);

    ELd = DebugTargetFrom(dest_is_secure);
    if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;
    else
        SS_bit = '0';

    return SS_bit;
```

Library pseudocode for shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// =====
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,  
// if it was not itself stepped.  
// Might return TRUE or FALSE if the previously executed instruction was an ISB or ERET executed  
// in the active-not-pending state, or if another exception was taken before the Software Step exception.  
// Returns FALSE otherwise, indicating that the previously executed instruction was executed in the  
// active-not-pending state, that is, the instruction was stepped.  
boolean SoftwareStep_DidNotStep();
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_SteppedEX

```
// Returns a value that describes the previously executed instruction. The result is valid only if  
// SoftwareStep_DidNotStep() returns FALSE.  
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX that failed its condition  
// Otherwise returns TRUE if the instruction was a Load-Exclusive class instruction, and FALSE if the  
// instruction was not a Load-Exclusive class instruction.  
boolean SoftwareStep_SteppedEX();
```

Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()  
// =====  
// Return CV and COND fields of instruction syndrome  
  
bits(5) ConditionSyndrome()  
  
    bits(5) syndrome;  
  
    if UsingAArch32() then  
        cond = AArch32.CurrentCond();  
        if PSTATE.T == '0' then // A32  
            syndrome<4> = '1';  
            // A conditional A32 instruction that is known to pass its condition code check  
            // can be presented either with COND set to 0xE, the value for unconditional, or  
            // the COND value held in the instruction.  
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then  
                syndrome<3:0> = '1110';  
            else  
                syndrome<3:0> = cond;  
        else // T32  
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:  
            // * CV set to 0 and COND is set to an UNKNOWN value  
            // * CV set to 1 and COND is set to the condition code for the condition that  
            // applied to the instruction.  
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then  
                syndrome<4> = '1';  
                syndrome<3:0> = cond;  
            else  
                syndrome<4> = '0';  
                syndrome<3:0> = bits(4) UNKNOWN;  
    else  
        syndrome<4> = '1';  
        syndrome<3:0> = '1110';  
  
    return syndrome;
```

Library pseudocode for shared/exceptions/exceptions/Exception

```
enumeration Exception {Exception_Uncategorized,      // Uncategorized or unknown reason
                        Exception_WFxTrap,           // Trapped WFI or WFE instruction
                        Exception_CP15RTTTrap,        // Trapped AArch32 MCR or MRC access, coproc=0b1111
                        Exception_CP15RRTTrap,        // Trapped AArch32 MCRR or MRRC access, coproc=0b11
                        Exception_CP14RTTTrap,        // Trapped AArch32 MCR or MRC access, coproc=0b1110
                        Exception_CP14DTTTrap,        // Trapped AArch32 LDC or STC access, coproc=0b1110
                        Exception_CP14RRTTrap,        // Trapped AArch32 MRRC access, coproc=0b1110
                        Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
                        Exception_FPIDTrap,           // Trapped access to SIMD or FP ID register
                        // Trapped BXJ instruction not supported in Armv8
                        Exception_LDST64BTrap,        // Trapped access to ST64BV, ST64BV0, ST64B and LD
                        // Trapped BXJ instruction not supported in Armv8
                        Exception_PACTrap,            // Trapped invalid PAC use
                        Exception_IllegalState,        // Illegal Execution state
                        Exception_SupervisorCall,      // Supervisor Call
                        Exception_HypervisorCall,      // Hypervisor Call
                        Exception_MonitorCall,         // Monitor Call or Trapped SMC instruction
                        Exception_SystemRegisterTrap,  // Trapped MRS or MSR system register access
                        Exception_ERetTrap,            // Trapped invalid ERET use
                        Exception_InstructionAbort,     // Instruction Abort or Prefetch Abort
                        Exception_PCAlignment,         // PC alignment fault
                        Exception_DataAbort,           // Data Abort
                        Exception_NV2DataAbort,        // Data abort at EL1 reported as being from EL2
                        Exception_PACFail,             // PAC Authentication failure
                        Exception_SPAlignment,         // SP alignment fault
                        Exception_FPTrappedException,  // IEEE trapped FP exception
                        Exception_SError,             // SError interrupt
                        Exception_Breakpoint,          // (Hardware) Breakpoint
                        Exception_SoftwareStep,        // Software Step
                        Exception_Watchpoint,          // Watchpoint
                        Exception_NV2Watchpoint,       // Watchpoint at EL1 reported as being from EL2
                        Exception_SoftwareBreakpoint,  // Software Breakpoint Instruction
                        Exception_VectorCatch,         // AArch32 Vector Catch
                        Exception_IRQ,                // IRQ interrupt
                        Exception_SVEAccessTrap,       // HCPTR trapped access to SVE
                        Exception_BranchTarget,        // Branch Target Identification
                        Exception_FIQ};               // FIQ interrupt
```

Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```
type ExceptionRecord is (
    Exception exceptype, // Exception class
    bits(25) syndrome,   // Syndrome record
    bits(5) syndrome2,   // ST64BV(0) return value register specifier
    bits(64) vaddress,   // Virtual fault address
    boolean ipavalid,    // Validity of Intermediate Physical fault address
    bit NS,              // Intermediate Physical fault address space
    bits(52) ipaddress)  // Intermediate Physical fault address
```

Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// =====
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)

    ExceptionRecord r;

    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.syndrome2 = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.NS = '0';
    r.ipaddress = Zeros();
    return r;
```

Library pseudocode for shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =====
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)
    bits(6) result;

    if level == -1 then
        assert Have52BitIPAAAndPASpaceExt();
        case statuscode of
            when Fault_AddressSize          result = '101001';
            when Fault_Translation           result = '101011';
            when Fault_SyncExternalOnWalk    result = '010011';
            when Fault_SyncParityOnWalk      result = '011011'; assert !HaveRASExt();
            otherwise                        Unreachable();

        return result;
    case statuscode of
        when Fault_AddressSize          result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag           result = '0010':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Permission            result = '0011':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Translation           result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal          result = '010000';
        when Fault_SyncExternalOnWalk    result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity            result = '011000';
        when Fault_SyncParityOnWalk      result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity          result = '011001';
        when Fault_AsyncExternal         result = '010001';
        when Fault_Alignment             result = '100001';
        when Fault_Debug                 result = '100010';
        when Fault_TLBConflict           result = '110000';
        when Fault_HWUpdateAccessFlag    result = '110001';
        when Fault_Lockdown              result = '110100'; // IMPLEMENTATION DEFINED
        when Fault_Exclusive             result = '110101'; // IMPLEMENTATION DEFINED
        otherwise                        Unreachable();

    return result;
```

Library pseudocode for shared/functions/aborts/IPAValid

```
// IPAValid()
// =====
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    if fault.s2fslwalk then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Permission,
            Fault_Translation,
            Fault_AddressSize
        };
    elsif fault.secondstage then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Translation,
            Fault_AddressSize
        };
    else
        return FALSE;
```

Library pseudocode for shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// =====

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```

Library pseudocode for shared/functions/aborts/IsDebugException

```
// IsDebugException()
// =====

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
```

Library pseudocode for shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =====
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk,
        Fault_AsyncExternal,
        Fault_AsyncParity
    });

// IsExternalAbort()
// =====

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode);
```

Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk
    });

// IsExternalSyncAbort()
// =====

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode);
```

Library pseudocode for shared/functions/aborts/IsFault

```
// IsFault()
// =====
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;

// IsFault()
// =====

boolean IsFault(Fault fault)
    return fault != Fault_None;

// IsFault()
// =====

boolean IsFault(PhysMemRetStatus retstatus)
    return retstatus.statuscode != Fault_None;
```


Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// =====
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// =====

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);
```

Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// =====

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
```

Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

```
// Returns the extended syndrome information for a second stage fault.
// <10> - Syndrome valid bit. The syndrome is only valid for certain types of access instruction.
// <9:8> - Access size.
// <7> - Sign extended (for loads).
// <6:2> - Transfer register.
// <1> - Transfer register is 64-bit.
// <0> - Instruction has acquire/release semantics.
bits(11) LSInstructionSyndrome();
```

Library pseudocode for shared/functions/cache/CACHE_OP

```
// CACHE_OP()
// =====
// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP(CacheRecord cache)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/cache/CacheOp

```
enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};
```

Library pseudocode for shared/functions/cache/CacheOpScope

```
enumeration CacheOpScope {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};
```

Library pseudocode for shared/functions/cache/CacheRecord

```
type CacheRecord is (
    AccType          acctype,          // Access type
    CacheOp          cacheop,          // Cache operation
    CacheOpScope     opscope,          // Cache operation type
    CacheType        cachetype,        // Cache type
    bits(64)         regval,
    FullAddress      paddress,
    bits(64)         vaddress,         // For VA operations
    integer          set,              // For SW operations
    integer          way,              // For SW operations
    integer          level,            // For SW operations
    Shareability     shareability,
    boolean          translated
);
```

Library pseudocode for shared/functions/cache/CacheType

```
enumeration CacheType {
    CacheType_Data,
    CacheType_Tag,
    CacheType_Data_Tag,
    CacheType_Instruction
};
```

Library pseudocode for shared/functions/cache/DCInstNeedsTranslation

```
// DCInstNeedsTranslation()
// =====
// Check whether Data Cache operation needs translation.

boolean DCInstNeedsTranslation(CacheOpScope opscope)
    if CLIDR_EL1.LoC == '000' then
        return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoC is before any

    if CLIDR_EL1.LoUU == '000' && opscope == CacheOpScope_PoU then
        return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoU is before any

    return TRUE;
```

Library pseudocode for shared/functions/cache/DecodeSW

```
// DecodeSW()
// =====
// Decode input value into set, way and level for SW instructions.

(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)
    level = UInt(regval[3:1]);
    (set, way, linesize) = GetCacheInfo(level, cachetype);
    return (set, way, level);
```

Library pseudocode for shared/functions/cache/GetCacheInfo

```
// Returns numsets, associativity & linesize.  
(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype);
```

Library pseudocode for shared/functions/cache/ICInstNeedsTranslation

```
// ICInstNeedsTranslation()  
// =====  
// Check whether Instruction Cache operation needs translation.  
  
boolean ICInstNeedsTranslation(CacheOpScope opscope)  
    return boolean IMPLEMENTATION_DEFINED "Instruction Cache needs translation";
```

Library pseudocode for shared/functions/common/ASR

```
// ASR()  
// =====  
  
bits(N) ASR(bits(N) x, integer shift)  
    assert shift >= 0;  
    if shift == 0 then  
        result = x;  
    else  
        (result, -) = ASR\_C(x, shift);  
    return result;
```

Library pseudocode for shared/functions/common/ASR_C

```
// ASR_C()  
// =====  
  
(bits(N), bit) ASR_C(bits(N) x, integer shift)  
    assert shift > 0;  
    extended_x = SignExtend(x, shift+N);  
    result = extended_x<shift+N-1:shift>;  
    carry_out = extended_x<shift-1>;  
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/Abs

```
// Abs()  
// =====  
  
integer Abs(integer x)  
    return if x >= 0 then x else -x;  
  
// Abs()  
// =====  
  
real Abs(real x)  
    return if x >= 0.0 then x else -x;
```

Library pseudocode for shared/functions/common/Align

```
// Align()  
// =====  
  
integer Align(integer x, integer y)  
    return y * (x DIV y);  
  
// Align()  
// =====  
  
bits(N) Align(bits(N) x, integer y)  
    return Align(UInt(x), y)<N-1:0>;
```

Library pseudocode for shared/functions/common/BitCount

```
// BitCount()
// =====

integer BitCount(bits(N) x)
  integer result = 0;
  for i = 0 to N-1
    if x<i> == '1' then
      result = result + 1;
  return result;
```

Library pseudocode for shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// =====

integer CountLeadingSignBits(bits(N) x)
  return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

Library pseudocode for shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// =====

integer CountLeadingZeroBits(bits(N) x)
  return N - (HighestSetBit(x) + 1);
```

Library pseudocode for shared/functions/common/Elem

```
// Elem[] - non-assignment form
// =====

bits(size) Elem(bits(N) vector, integer e, integer size)
  assert e >= 0 && (e+1)*size <= N;
  return vector<e*size+size-1 : e*size>;

// Elem[] - non-assignment form
// =====

bits(size) Elem(bits(N) vector, integer e)
  return Elem[vector, e, size];

// Elem[] - assignment form
// =====

Elem(bits(N) &vector, integer e, integer size) = bits(size) value
  assert e >= 0 && (e+1)*size <= N;
  vector<(e+1)*size-1:e*size> = value;
  return;

// Elem[] - assignment form
// =====

Elem(bits(N) &vector, integer e) = bits(size) value
  Elem[vector, e, size] = value;
  return;
```

Library pseudocode for shared/functions/common/Extend

```
// Extend()
// =====

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

// Extend()
// =====

bits(N) Extend(bits(M) x, boolean unsigned)
    return Extend(x, N, unsigned);
```

Library pseudocode for shared/functions/common/HighestSetBit

```
// HighestSetBit()
// =====

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

Library pseudocode for shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

Library pseudocode for shared/functions/common/IsOnes

```
// IsOnes()
// =====

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

Library pseudocode for shared/functions/common/IsZero

```
// IsZero()
// =====

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

Library pseudocode for shared/functions/common/IsZeroBit

```
// IsZeroBit()
// =====

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

Library pseudocode for shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/LSL_C

```
// LSL_C()
// =====

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/LSR_C

```
// LSR_C()
// =====

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/LowestSetBit

```
// LowestSetBit()
// =====

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

Library pseudocode for shared/functions/common/Max

```
// Max()
// =====

integer Max(integer a, integer b)
    return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
    return if a >= b then a else b;
```

Library pseudocode for shared/functions/common/Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```

Library pseudocode for shared/functions/common/Ones

```
// Ones()
// =====

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// =====

bits(N) Ones()
    return Ones(N);
```

Library pseudocode for shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR\_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/ROR_C

```
// ROR_C()
// =====

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/Replicate

```
// Replicate()
// =====

bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);
```

Library pseudocode for shared/functions/common/RoundDown

```
integer RoundDown(real x);
```

Library pseudocode for shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// =====

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

Library pseudocode for shared/functions/common/RoundUp

```
integer RoundUp(real x);
```

Library pseudocode for shared/functions/common/SInt

```
// SInt()
// =====

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

Library pseudocode for shared/functions/common/SignExtend

```
// SignExtend()
// =====

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;

// SignExtend()
// =====

bits(N) SignExtend(bits(M) x)
    return SignExtend(x, N);
```

Library pseudocode for shared/functions/common/UInt

```
// UInt()
// =====

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```


Library pseudocode for shared/functions/common/ZeroExtend

```
// ZeroExtend()
// =====

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;

// ZeroExtend()
// =====

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =====

bits(N) Zeros(integer N)
    return Replicate('0',N);

// Zeros()
// =====

bits(N) Zeros()
    return Zeros(N);
```

Library pseudocode for shared/functions/counters/GenericCounterTick

```
// GenericCounterTick()
// =====
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
    if CNTCR.EN == '0' then
        return;
    if HaveCNTSCExt() && CNTCR.SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR);
    else
        PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
```

Library pseudocode for shared/functions/counters/PhysicalCount

```
bits(88) PhysicalCount;
```

Library pseudocode for shared/functions/crc/BitReverse

```
// BitReverse()
// =====

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<N-i-1> = data<i>;
    return result;
```

Library pseudocode for shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// =====

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

Library pseudocode for shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// =====

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
  assert N > 32;
  for i = N-1 downto 32
    if data<i> == '1' then
      data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
  return data<31:0>;
```

Library pseudocode for shared/functions/crypto/AESInvMixColumns

```
// AESInvMixColumns()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
  bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
  bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
  bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
  bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

  bits(4*8) out0;
  bits(4*8) out1;
  bits(4*8) out2;
  bits(4*8) out3;

  for c = 0 to 3
    out0<c*8+:8> = Ffmul0E(in0<c*8+:8>) EOR Ffmul0B(in1<c*8+:8>) EOR Ffmul0D(in2<c*8+:8>) EOR Ffmul0A(in3<c*8+:8>);
    out1<c*8+:8> = Ffmul09(in0<c*8+:8>) EOR Ffmul0E(in1<c*8+:8>) EOR Ffmul0B(in2<c*8+:8>) EOR Ffmul0A(in3<c*8+:8>);
    out2<c*8+:8> = Ffmul0D(in0<c*8+:8>) EOR Ffmul09(in1<c*8+:8>) EOR Ffmul0E(in2<c*8+:8>) EOR Ffmul0B(in3<c*8+:8>);
    out3<c*8+:8> = Ffmul0B(in0<c*8+:8>) EOR Ffmul0D(in1<c*8+:8>) EOR Ffmul09(in2<c*8+:8>) EOR Ffmul0E(in3<c*8+:8>);

  return (
    out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
    out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
    out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
    out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
  );
```

Library pseudocode for shared/functions/crypto/AESInvShiftRows

```
// AESInvShiftRows()
// =====
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
  return (
    op< 24+:8> : op< 48+:8> : op< 72+:8> : op< 96+:8> :
    op<120+:8> : op< 16+:8> : op< 40+:8> : op< 64+:8> :
    op< 88+:8> : op<112+:8> : op<  8+:8> : op< 32+:8> :
    op< 56+:8> : op< 80+:8> : op<104+:8> : op<  0+:8>
  );
```

Library pseudocode for shared/functions/crypto/AESInvSubBytes

```
// AESInvSubBytes()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
  // Inverse S-box values
  bits(16*16*8) GF2_inv = (
    /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
    /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
    /*E*/ 0x619953833cbbefbc8b0f52aae4d3be0a0<127:0> :
    /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
    /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
    /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
    /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
    /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
    /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
    /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :
    /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
    /*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
    /*4*/ 0x92b6655dcc5ca4d41698688664f6f872<127:0> :
    /*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
    /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
    /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c<127:0> :
    /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0>
  );
  bits(128) out;
  for i = 0 to 15
    out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
  return out;
```

Library pseudocode for shared/functions/crypto/AESMixColumns

```
// AESMixColumns()
// =====
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
  bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
  bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
  bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
  bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

  bits(4*8) out0;
  bits(4*8) out1;
  bits(4*8) out2;
  bits(4*8) out3;

  for c = 0 to 3
    out0<c*8+:8> = FFMul02(in0<c*8+:8>) EOR FFMul03(in1<c*8+:8>) EOR          in2<c*8+:8> EOR
    out1<c*8+:8> =          in0<c*8+:8> EOR FFMul02(in1<c*8+:8>) EOR FFMul03(in2<c*8+:8>) EOR
    out2<c*8+:8> =          in0<c*8+:8> EOR          in1<c*8+:8> EOR FFMul02(in2<c*8+:8>) EOR FFMul03(in3<c*8+:8>)
    out3<c*8+:8> = FFMul03(in0<c*8+:8>) EOR          in1<c*8+:8> EOR          in2<c*8+:8> EOR FFMul02(in3<c*8+:8>)

  return (
    out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
    out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
    out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
    out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
  );
```

Library pseudocode for shared/functions/crypto/AESShiftRows

```
// AESShiftRows()
// =====
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
    return (
        op< 88+:8> : op< 48+:8> : op<  8+:8> : op< 96+:8> :
        op< 56+:8> : op< 16+:8> : op<104+:8> : op< 64+:8> :
        op< 24+:8> : op<112+:8> : op< 72+:8> : op< 32+:8> :
        op<120+:8> : op< 80+:8> : op< 40+:8> : op<  0+:8>
    );
```

Library pseudocode for shared/functions/crypto/AESSubBytes

```
// AESSubBytes()
// =====
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    bits(16*16*8) GF2 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt>(op<i*8+:8>)*8+:8>;
    return out;
```

Library pseudocode for shared/functions/crypto/FFmul02

```
// FFmul02()
// =====

bits(8) FFmul02(bits(8) b)
    bits(256*8) FFmul_02 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAF9ABB5B7B1B3BDBFB9BB<127:0> :
        /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
        /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
        /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
        /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCAC8C6C4C2C0<127:0> :
        /*5*/ 0xBEBBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
        /*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
    );
    return FFmul_02<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul03

```
// FFmul03()
// =====

bits(8) FFmul03(bits(8) b)
    bits(256*8) FFmul_03 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/ 0x2A292C2F26252023323134373E3D383B<127:0> :
        /*D*/ 0x7A797C7F76757073626164676E6D686B<127:0> :
        /*C*/ 0x4A494C4F46454043525154575E5D585B<127:0> :
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/ 0x8A898C8F86858083929194979E9D989B<127:0> :
        /*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
        /*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
        /*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
        /*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
    );
    return FFmul_03<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul09

```
// FFmul09()
// =====

bits(8) FFmul09(bits(8) b)
    bits(256*8) FFmul_09 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
        /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
        /*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
        /*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
        /*6*/ 0x3A3328211E170C05727B6069565F444D<127:0> :
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :
        /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
        /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
        /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
    );
    return FFmul_09<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0B

```
// FFmul0B()
// =====

bits(8) FFmul0B(bits(8) b)
    bits(256*8) FFmul_0B = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
        /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
        /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
        /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
        /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
        /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
        /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
        /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
        /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
    );
    return FFmul_0B<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =====

bits(8) FFmul0D(bits(8) b)
    bits(256*8) FFmul_0D = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBCECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADD00<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0E

```
// FFmul0E()
// =====

bits(8) FFmul0E(bits(8) b)
    bits(256*8) FFmul_0E = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
        /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
        /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
        /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
        /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDFAFA1<127:0> :
        /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
        /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
        /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
        /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
        /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
    );
    return FFmul_0E<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/HaveAESExt

```
// HaveAESExt()
// =====
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveBit128PMULLExt

```
// HaveBit128PMULLExt()
// =====
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA1Ext

```
// HaveSHA1Ext()
// =====
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA256Ext

```
// HaveSHA256Ext()
// =====
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA3Ext

```
// HaveSHA3Ext()
// =====
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    if !HasArchVersion\(ARMv8p2\) || !(HaveSHA1Ext\(\) && HaveSHA256Ext\(\)) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA512Ext

```
// HaveSHA512Ext()
// =====
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    if !HasArchVersion\(ARMv8p2\) || !(HaveSHA1Ext\(\) && HaveSHA256Ext\(\)) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
```


Library pseudocode for shared/functions/crypto/HaveSM3Ext

```
// HaveSM3Ext()
// =====
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSM4Ext

```
// HaveSM4Ext()
// =====
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

Library pseudocode for shared/functions/crypto/SHA256hash

```
// SHA256hash()
// =====

bits(128) SHA256hash(bits (128) X, bits(128) Y, bits(128) W, boolean part1)
    bits(32) chs, maj, t;

    for e = 0 to 3
        chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
        maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
        t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
        X<127:96> = t + X<127:96>;
        Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
        <Y, X> = ROL(Y : X, 32);
    return (if part1 then X else Y);
```

Library pseudocode for shared/functions/crypto/SHAchoose

```
// SHAchoose()
// =====

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return ((y EOR z) AND x) EOR z;
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()  
// =====  
  
bits(32) SHAhashSIGMA0(bits(32) x)  
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()  
// =====  
  
bits(32) SHAhashSIGMA1(bits(32) x)  
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

Library pseudocode for shared/functions/crypto/SHAmajority

```
// SHAmajority()  
// =====  
  
bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)  
    return ((x AND y) OR ((x OR y) AND z));
```

Library pseudocode for shared/functions/crypto/SHAparity

```
// SHAparity()  
// =====  
  
bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)  
    return (x EOR y EOR z);
```

Library pseudocode for shared/functions/crypto/Sbox

```
// Sbox()  
// =====  
// Used in SM4E crypto instruction  
  
bits(8) Sbox(bits(8) sboxin)  
    bits(8) sboxout;  
    bits(2048) sboxstring = 0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491e  
  
    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;  
    return sboxout;
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they  
// record any part of the physical address region of size bytes starting at paddress.  
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid  
// is also cleared if it records any part of the address region.  
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusives monitor for the specified processorid.  
ClearExclusiveLocal(integer processorid);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// =====
// Clear the local Exclusives monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

```
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();
```

Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

```
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at address.
boolean IsExclusiveGlobal(FullAddress address, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

```
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at address.
boolean IsExclusiveLocal(FullAddress address, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at address in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress address, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at address in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress address, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// =====

boolean AArch32.HaveHPDExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// =====

boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/Have52BitIPAAndPASpaceExt

```
// Have52BitIPAAndPASpaceExt()
// =====
// Returns TRUE if 52-bit IPA and PA extension support
// is implemented, and FALSE otherwise.

boolean Have52BitIPAAndPASpaceExt()
    return (HasArchVersion(ARMv8p7) &&
        boolean IMPLEMENTATION_DEFINED "Has 52-bit IPA and PA support" &&
        Have52BitVAExt() && Have52BitPAExt());
```

Library pseudocode for shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// =====
// Returns TRUE if Large Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has large 52-bit PA/IPA support";
```

Library pseudocode for shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// =====
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has large 52-bit VA support";
```

Library pseudocode for shared/functions/extension/HaveAArch32BF16Ext

```
// HaveAArch32BF16Ext()
// =====
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch32 BFloat16 extension";
```

Library pseudocode for shared/functions/extension/HaveAArch32Int8MatMulExt

```
// HaveAArch32Int8MatMulExt()
// =====
// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support
// is implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch32 Int8 Mat Mul extension";
```

Library pseudocode for shared/functions/extension/HaveAltFP

```
// HaveAltFP()
// =====
// Returns TRUE if alternative Floating-point extension support
// is implemented, and FALSE otherwise.

boolean HaveAltFP()
    return HasArchVersion(ARMv8p7);
```

Library pseudocode for shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// =====

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveBF16Ext

```
// HaveBF16Ext()
// =====
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveBF16Ext()
    return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch64 BF16 extension");
```

Library pseudocode for shared/functions/extension/HaveBTIExt

```
// HaveBTIExt()
// =====
// Returns TRUE if support for Branch Target Identification is implemented.

boolean HaveBTIExt()
    return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveBlockBBM

```
// HaveBlockBBM()
// =====
// Returns TRUE if support for changing block size without requiring break-before-make is implemented.

boolean HaveBlockBBM()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveCNTSCEExt

```
// HaveCNTSCEExt()
// =====
// Returns TRUE if the Generic Counter Scaling is implemented, and FALSE
// otherwise.

boolean HaveCNTSCEExt()
    return (HasArchVersion(ARMv8p4) &&
            boolean IMPLEMENTATION_DEFINED "Has Generic Counter Scaling support");
```

Library pseudocode for shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// =====

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveDGHEExt

```
// HaveDGHEExt()
// =====
// Returns TRUE if Data Gathering Hint instruction support is implemented, and FALSE otherwise.

boolean HaveDGHEExt()
    return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";
```

Library pseudocode for shared/functions/extension/HaveDITExt

```
// HaveDITExt()  
// =====  
  
boolean HaveDITExt()  
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()  
// =====  
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.  
  
boolean HaveDOTPExt()  
    return HasArchVersion(ARMv8p4) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has Dot Product feature support");
```

Library pseudocode for shared/functions/extension/HaveDoPD

```
// HaveDoPD()  
// =====  
// Returns TRUE if Debug Over Power Down extension  
// support is implemented and FALSE otherwise.  
  
boolean HaveDoPD()  
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has DoPD extension";
```

Library pseudocode for shared/functions/extension/HaveDoubleFaultExt

```
// HaveDoubleFaultExt()  
// =====  
  
boolean HaveDoubleFaultExt()  
    return (HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESB());
```

Library pseudocode for shared/functions/extension/HaveDoubleLock

```
// HaveDoubleLock()  
// =====  
// Returns TRUE if support for the OS Double Lock is implemented.  
  
boolean HaveDoubleLock()  
    return !HasArchVersion(ARMv8p4) || boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented";
```

Library pseudocode for shared/functions/extension/HaveE0PDExt

```
// HaveE0PDExt()  
// =====  
// Returns TRUE if support for constant fault times for unprivileged accesses  
// to the memory map is implemented.  
  
boolean HaveE0PDExt()  
    return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveECVExt

```
// HaveECVExt()  
// =====  
// Returns TRUE if Enhanced Counter Virtualization extension  
// support is implemented, and FALSE otherwise.  
  
boolean HaveECVExt()  
    return HasArchVersion(ARMv8p6);
```

Library pseudocode for shared/functions/extension/HaveEMPAMExt

```
// HaveEMPAMExt()  
// =====  
// Returns TRUE if Enhanced MPAM is implemented, and FALSE otherwise.  
  
boolean HaveEMPAMExt()  
    return (HasArchVersion(ARMv8p6) &&  
            HaveMPAMExt()) &&  
            boolean IMPLEMENTATION_DEFINED "Has enhanced MPAM extension");
```

Library pseudocode for shared/functions/extension/HaveExtendedCacheSets

```
// HaveExtendedCacheSets()  
// =====  
  
boolean HaveExtendedCacheSets()  
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveExtendedECDebugEvents

```
// HaveExtendedECDebugEvents()  
// =====  
  
boolean HaveExtendedECDebugEvents()  
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveExtendedExecuteNeverExt

```
// HaveExtendedExecuteNeverExt()  
// =====  
  
boolean HaveExtendedExecuteNeverExt()  
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveFCADDExt

```
// HaveFCADDExt()  
// =====  
  
boolean HaveFCADDExt()  
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveFGTExt

```
// HaveFGTExt()  
// =====  
// Returns TRUE if Fine Grained Trap is implemented, and FALSE otherwise.  
  
boolean HaveFGTExt()  
    return HasArchVersion(ARMv8p6) && !ELUsingAArch32(EL2);
```

Library pseudocode for shared/functions/extension/HaveFJCVTZSExt

```
// HaveFJCVTZSExt()  
// =====  
  
boolean HaveFJCVTZSExt()  
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```
// HaveFP16MulNoRoundingToFP32Ext()
// =====
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate to FP32 instructions,
// and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
    if !HaveFP16Ext() then return FALSE;
    if HasArchVersion(ARMv8p4) then return TRUE;
    return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");
```

Library pseudocode for shared/functions/extension/HaveFeatHCX

```
// HaveFeatHCX()
// =====
// Returns TRUE if HCRX_EL2 Trap Control register is implemented,
// and FALSE otherwise.

boolean HaveFeatHCX()
    return HasArchVersion(ARMv8p7);
```

Library pseudocode for shared/functions/extension/HaveFeatLS64

```
// HaveFeatLS64()
// =====
// Returns TRUE if the LD64B, ST64B, ST64BV, and ST64BV0 instructions are
// supported, and FALSE otherwise.

boolean HaveFeatLS64()
    return (HasArchVersion(ARMv8p7) &&
        boolean IMPLEMENTATION_DEFINED "Has Load Store 64-Byte instruction support");
```

Library pseudocode for shared/functions/extension/HaveFeatRPRES

```
// HaveFeatRPRES()
// =====
// Returns TRUE if reciprocal estimate implements 12-bit precision
// when FPCR.AH=1, and FALSE otherwise.

boolean HaveFeatRPRES()
    return (HasArchVersion(ARMv8p7) &&
        (boolean IMPLEMENTATION_DEFINED "Has increased Reciprocal Estimate and Square Root Estimate precision")
        && HaveAltFP());
```

Library pseudocode for shared/functions/extension/HaveFeatWFXt

```
// HaveFeatWFXt()
// =====
// Returns TRUE if WFET and WFIT instruction support is implemented,
// and FALSE otherwise.

boolean HaveFeatWFXt()
    return HasArchVersion(ARMv8p7);
```

Library pseudocode for shared/functions/extension/HaveFeatWFXt2

```
// HaveFeatWFXt2()
// =====
// Returns TRUE if the register number is reported in the ESR_ELx
// on exceptions to WFIT and WFET.

boolean HaveFeatWFXt2()
    return HaveFeatWFXt() && boolean IMPLEMENTATION_DEFINED "Has feature WFXt2";
```


Library pseudocode for shared/functions/extension/HaveFeatXS

```
// HaveFeatXS()  
// =====  
// Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier  
// are supported, and FALSE otherwise.  
  
boolean HaveFeatXS()  
    return HasArchVersion\(ARMv8p7\);
```

Library pseudocode for shared/functions/extension/HaveFlagFormatExt

```
// HaveFlagFormatExt()  
// =====  
// Returns TRUE if flag format conversion instructions implemented.  
  
boolean HaveFlagFormatExt()  
    return HasArchVersion\(ARMv8p5\);
```

Library pseudocode for shared/functions/extension/HaveFlagManipulateExt

```
// HaveFlagManipulateExt()  
// =====  
// Returns TRUE if flag manipulate instructions are implemented.  
  
boolean HaveFlagManipulateExt()  
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveFrintExt

```
// HaveFrintExt()  
// =====  
// Returns TRUE if FRINT instructions are implemented.  
  
boolean HaveFrintExt()  
    return HasArchVersion\(ARMv8p5\);
```

Library pseudocode for shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()  
// =====  
  
boolean HaveHPMDExt()  
    return HasArchVersion\(ARMv8p1\);
```

Library pseudocode for shared/functions/extension/HaveIDSExt

```
// HaveIDSExt()  
// =====  
// Returns TRUE if ID register handling feature is implemented.  
  
boolean HaveIDSExt()  
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveIESB

```
// HaveIESB()  
// =====  
  
boolean HaveIESB()  
    return (HaveRASExt\(\) &&  
           boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");
```

Library pseudocode for shared/functions/extension/HaveInt8MatMulExt

```
// HaveInt8MatMulExt()
// =====
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveInt8MatMulExt()
    return HasArchVersion\(ARMv8p6\) || (HasArchVersion\(ARMv8p2\) && boolean IMPLEMENTATION_DEFINED "Has AA
```

Library pseudocode for shared/functions/extension/HaveLSE2Ext

```
// HaveLSE2Ext()
// =====
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.

boolean HaveLSE2Ext()
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveMPAMExt

```
// HaveMPAMExt()
// =====
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveMPAMExt()
    return (HasArchVersion\(ARMv8p2\) &&
        boolean IMPLEMENTATION_DEFINED "Has MPAM extension");
```

Library pseudocode for shared/functions/extension/HaveMTE2Ext

```
// HaveMTE2Ext()
// =====
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.

boolean HaveMTE2Ext()
    if !HasArchVersion\(ARMv8p5\) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE2 extension";
```

Library pseudocode for shared/functions/extension/HaveMTE3Ext

```
// HaveMTE3Ext()
// =====
// Returns TRUE if MTE Asymmetric Fault Handling support is
// implemented, and FALSE otherwise.

boolean HaveMTE3Ext()
    return ((HasArchVersion\(ARMv8p7\) && HaveMTE2Ext\(\)) || (HasArchVersion\(ARMv8p5\) &&
        boolean IMPLEMENTATION_DEFINED "Has MTE3 extension"));
```

Library pseudocode for shared/functions/extension/HaveMTEExt

```
// HaveMTEExt()
// =====
// Returns TRUE if MTE implemented, and FALSE otherwise.

boolean HaveMTEExt()
    if !HasArchVersion\(ARMv8p5\) then
        return FALSE;
    if HaveMTE2Ext\(\) then
        return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE extension";
```

Library pseudocode for shared/functions/extension/HaveNV2Ext

```
// HaveNV2Ext()
// =====
// Returns TRUE if Enhanced Nested Virtualization is implemented.

boolean HaveNV2Ext()
    return (HasArchVersion(ARMv8p4) && HaveNVExt()
        && boolean IMPLEMENTATION_DEFINED "Has support for Enhanced Nested Virtualization");
```

Library pseudocode for shared/functions/extension/HaveNVExt

```
// HaveNVExt()
// =====
// Returns TRUE if Nested Virtualization is implemented.

boolean HaveNVExt()
    return HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has Nested Virtualization";
```

Library pseudocode for shared/functions/extension/HaveNoSecurePMUDisableOverride

```
// HaveNoSecurePMUDisableOverride()
// =====

boolean HaveNoSecurePMUDisableOverride()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveNoninvasiveDebugAuth

```
// HaveNoninvasiveDebugAuth()
// =====
// Returns TRUE if the Non-invasive debug controls are implemented.

boolean HaveNoninvasiveDebugAuth()
    return !HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HavePAN3Ext

```
// HavePAN3Ext()
// =====
// Returns TRUE if SCTLR_EL1.EPAN and SCTLR_EL2.EPAN support is implemented,
// and FALSE otherwise.

boolean HavePAN3Ext()
    return HasArchVersion(ARMv8p7) || (HasArchVersion(ARMv8p1) &&
        boolean IMPLEMENTATION_DEFINED "Has PAN3 extension");
```

Library pseudocode for shared/functions/extension/HavePANExt

```
// HavePANExt()
// =====

boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HavePMUv3p7

```
// HavePMUv3p7()
// =====
// Returns TRUE if the PMUv3p7 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p7()
    return (HasArchVersion(ARMv8p7) && Havev85PMU() &&
        boolean IMPLEMENTATION_DEFINED "Has PMUv3p7 extension");
```

Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()
// =====

boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HavePrivATExt

```
// HavePrivATExt()
// =====

boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveQRDMLAExt

```
// HaveQRDMLAExt()
// =====

boolean HaveQRDMLAExt()
    return HasArchVersion(ARMv8p1);

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveRASExt

```
// HaveRASExt()
// =====

boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) ||
            boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

Library pseudocode for shared/functions/extension/HaveRNG

```
// HaveRNG()
// =====
// Returns TRUE if Random Number Generator extension
// support is implemented and FALSE otherwise.

boolean HaveRNG()
    return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has RNG extension";
```

Library pseudocode for shared/functions/extension/HaveSBExt

```
// HaveSBExt()
// =====
// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveSBExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension";
```

Library pseudocode for shared/functions/extension/HaveSSBSExt

```
// HaveSSBSExt()
// =====
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveSSBSExt()
    return HasArchVersion\(ARMv8p5\) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension";
```

Library pseudocode for shared/functions/extension/HaveSecureEL2Ext

```
// HaveSecureEL2Ext()
// =====
// Returns TRUE if Secure EL2 is implemented.

boolean HaveSecureEL2Ext()
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveSecureExtDebugView

```
// HaveSecureExtDebugView()
// =====
// Returns TRUE if support for Secure and Non-secure views of debug peripherals is implemented.

boolean HaveSecureExtDebugView()
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveSelfHostedTrace

```
// HaveSelfHostedTrace()
// =====

boolean HaveSelfHostedTrace()
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveSmallTranslationTblExt

```
// HaveSmallTranslationTblExt()
// =====
// Returns TRUE if Small Translation Table Support is implemented.

boolean HaveSmallTranslationTableExt()
    return HasArchVersion\(ARMv8p4\) && boolean IMPLEMENTATION_DEFINED "Has Small Translation Table extension";
```

Library pseudocode for shared/functions/extension/HaveSoftwareLock

```
// HaveSoftwareLock()
// =====
// Returns TRUE if Software Lock is implemented.

boolean HaveSoftwareLock(Component component)
    if Havev8p4Debug\(\) then
        return FALSE;
    if HaveDoPD\(\) && component != Component\_CTI then
        return FALSE;
    case component of
        when Component\_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
        when Component\_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
        when Component\_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
        otherwise
            Unreachable\(\);
```

Library pseudocode for shared/functions/extension/HaveStage2MemAttrControl

```
// HaveStage2MemAttrControl()
// =====
// Returns TRUE if support for Stage2 control of memory types and cacheability attributes is implemented.

boolean HaveStage2MemAttrControl()
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfiling

```
// HaveStatisticalProfiling()
// =====
// Returns TRUE if Statistical Profiling Extension is implemented,
// and FALSE otherwise.

boolean HaveStatisticalProfiling()
    return HasArchVersion\(ARMv8p2\);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p1

```
// HaveStatisticalProfilingv1p1()
// =====
// Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p1()
    return (HasArchVersion\(ARMv8p3\) &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p1 extension");
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p2

```
// HaveStatisticalProfilingv1p2()
// =====
// Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p2()
    return (HasArchVersion\(ARMv8p7\) && HaveStatisticalProfiling\(\) &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p2 extension");
```

Library pseudocode for shared/functions/extension/HaveTWEDEExt

```
// HaveTWEDEExt()
// =====
// Returns TRUE if Delayed Trapping of WFE instruction support is implemented, and FALSE otherwise.

boolean HaveTWEDEExt()
    return boolean IMPLEMENTATION_DEFINED "Has TWED extension";
```

Library pseudocode for shared/functions/extension/HaveTraceExt

```
// HaveTraceExt()
// =====
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";
```

Library pseudocode for shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```
// HaveTrapLoadStoreMultipleDeviceExt()
// =====

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion\(ARMv8p2\);
```

Library pseudocode for shared/functions/extension/HaveUAOExt

```
// HaveUAOExt()  
// =====  
  
boolean HaveUAOExt()  
    return HasArchVersion\(ARMv8p2\);
```

Library pseudocode for shared/functions/extension/HaveV82Debug

```
// HaveV82Debug()  
// =====  
  
boolean HaveV82Debug()  
    return HasArchVersion\(ARMv8p2\);
```

Library pseudocode for shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt()  
// =====  
  
boolean HaveVirtHostExt()  
    return HasArchVersion\(ARMv8p1\);
```

Library pseudocode for shared/functions/extension/Havev85PMU

```
// Havev85PMU()  
// =====  
// Returns TRUE if v8.5-Performance Monitor Unit extension  
// support is implemented, and FALSE otherwise.  
  
boolean Havev85PMU()  
    return HasArchVersion\(ARMv8p5\) && boolean IMPLEMENTATION_DEFINED "Has PMUv3p5 extension";
```

Library pseudocode for shared/functions/extension/Havev8p4Debug

```
// Havev8p4Debug()  
// =====  
// Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.  
  
boolean Havev8p4Debug()  
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/InsertIESBBeforeException

```
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable  
// SError interrupt must be taken before executing any instructions in the exception handler.  
// However, this can be before the branch to the exception handler is made.  
boolean InsertIESBBeforeException(bits(2) el);
```

Library pseudocode for shared/functions/externalaborts/HandleExternalAbort

```
// HandleExternalAbort()
// =====
// Takes a Synchronous/Asynchronous abort based on fault.

HandleExternalAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                    AddressDescriptor memaddrdesc, integer size,
                    AccessDescriptor accdesc)
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
            (!HaveRASExt() && memretstatus.statuscode IN {Fault_SyncParity,
                                                         Fault_AsyncParity}));

    fault = NoFault();
    fault.statuscode = memretstatus.statuscode;
    fault.write = iswrite;
    fault.extflag = memretstatus.extflag;
    fault.acctype = memretstatus.acctype;
    // It is implementation specific whether external aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) &&
        !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                             size, accdesc)) then
        if fault.statuscode == Fault_SyncParity then
            fault.statuscode = Fault_AsyncParity;
        else
            fault.statuscode = Fault_AsyncExternal;

    if HaveRASExt() then
        fault.errortype = PEErrState(memretstatus);
    else
        fault.errortype = bits(2) UNKNOWN;

    if IsExternalSyncAbort(fault) then
        if UsingAArch32() then
            AArch32.Abort(memaddrdesc.vaddress<31:0>, fault);
        else
            AArch64.Abort(memaddrdesc.vaddress, fault);
    else
        PendSErrorInterrupt(fault);
```

Library pseudocode for shared/functions/externalaborts/HandleExternalReadAbort

```
// HandleExternalReadAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

HandleExternalReadAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                        integer size, AccessDescriptor accdesc)
    iswrite = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```


Library pseudocode for shared/functions/externalaborts/HandleExternalTTWAbort

```
// HandleExternalTTWAbort()
// =====
// Take Asynchronous abort or update FaultRecord for Translation Walk
// based on PhysMemRetStatus.

FaultRecord HandleExternalTTWAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                                   AddressDescriptor memaddrdesc,
                                   AccessDescriptor accdesc, integer size,
                                   FaultRecord input_fault)
    output_fault = input_fault;
    output_fault.extflag = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (IsExternalSyncAbort(output_fault) &&
        !IsExternalAbortTakenSynchronously(memretstatus, iswrite,
                                           memaddrdesc,
                                           size, accdesc)) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_AsyncParity;
        else
            output_fault.statuscode = Fault_AsyncExternal;

    // If a synchronous fault is on a translation table walk, then update
    // the fault type
    if IsExternalSyncAbort(output_fault) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault_SyncExternalOnWalk;
    if HaveRASExt() then
        output_fault.errortype = PEErrrorState(memretstatus);
    else
        output_fault.errortype = bits(2) UNKNOWN;
    if !IsExternalSyncAbort(output_fault) then
        PendSErrorInterrupt(output_fault);
        output_fault.statuscode = Fault_None;
    return output_fault;
```

Library pseudocode for shared/functions/externalaborts/HandleExternalWriteAbort

```
// HandleExternalWriteAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

HandleExternalWriteAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                        integer size, AccessDescriptor accdesc)
    iswrite = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

Library pseudocode for shared/functions/externalaborts/IsExternalAbortTakenSynchronously

```
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.
boolean IsExternalAbortTakenSynchronously(PhysMemRetStatus memstatus,
                                         boolean iswrite,
                                         AddressDescriptor desc,
                                         integer size,
                                         AccessDescriptor accdesc);
```

Library pseudocode for shared/functions/externalaborts/PEErrorState

```
// Return the implementation specific PE error state.
// memstatus is the response returned from the system.
// It is implementation specific whether this is used or ignored.
bits(2) PEErrorState(PhysMemRetStatus memstatus);
```

Library pseudocode for shared/functions/externalaborts/PendSErrorInterrupt

```
// Pend the SError.
PendSErrorInterrupt(FaultRecord fault);
```

Library pseudocode for shared/functions/float/bfloat/BFAdd

```
// BFAdd()
// =====
// Single-precision add following BFloat16 computation behaviors.

bits(32) BFAdd(bits(32) op1, bits(32) op2)

    bits(32) result;

    FPCRTyp fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPTyp_QNaN || type2 == FPTyp_QNaN then
        result = FPDefaultNaN(fpcr);
    else
        inf1 = (type1 == FPTyp_Infinity);
        inf2 = (type2 == FPTyp_Infinity);
        zero1 = (type1 == FPTyp_Zero);
        zero2 = (type2 == FPTyp_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0'); // Positive sign when Round to Odd
            else
                result = BFRound(result_value);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFDotAdd

```
// BFDotAdd()
// =====
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

bits(32) BFDotAdd(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
                  bits(16) op2_a, bits(16) op2_b, FPCRTyp fpcr)

    bits(32) prod;

    prod = BFAdd(BFMul(op1_a, op2_a), BFMul(op1_b, op2_b));
    result = BFAdd(addend, prod);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFMatMulAdd

```
// BFMatMulAdd()
// =====
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)

    assert N == 128;

    bits(N) result;
    bits(32) sum;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 1
                bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
                bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
                bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
                bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
                sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFMul

```
// BFMul()
// =====
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(32) BFMul(bits(16) op1, bits(16) op2)

    bits(32) result;

    FPCRType fpcr = FPCR[];
    (type1, sign1, value1) = BFUnpack(op1);
    (type2, sign2, value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN(fpcr);
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr);
        elsif inf1 || inf2 then
            result = FPIInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = BFRound(value1*value2);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFMulAdd

```
// BFMulAdd()
// =====
// Used by BFMLALB and BFMLALT instructions.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean altfp = HaveAltFP() && fpcr.AH == '1'; // When TRUE:
    boolean fpexc = !altfp;                       // Do not generate floating point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';            // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00';              // Use RNE rounding mode
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/bfloat/BFNeg

```
// BFNeg()
// =====

bits(16) BFNeg(bits(16) op)
    return NOT(op<15>) : op<14:0>;
```

Library pseudocode for shared/functions/float/bfloat/BFRound

```
// BFRound()
// =====
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(32) BFRound(real op)

    assert op != 0.0;
    bits(32) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126;  E = 8;  F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign);  // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFUnpack

```
// BFUnpack()
// =====
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)

    assert N IN {16,32};

    if N == 16 then
        sign    = fpval<15>;
        exp     = fpval<14:7>;
        frac    = fpval<6:0> : Zeros(16);
    else // N == 32
        sign    = fpval<31>;
        exp     = fpval<30:23>;
        frac    = fpval<22:0>;

    if IsZero(exp) then
        fptype = FPType_Zero; value = 0.0;    // Fixed Flush to Zero
    elseif IsOnes(exp) then
        if IsZero(frac) then
            fptype = FPType_Infinity; value = 2.0^1000000;
        else // no SNaN for BF16 arithmetic
            fptype = FPType_QNaN; value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

Library pseudocode for shared/functions/float/bfloat/FPConvertBF

```
// FPConvertBF()
// =====
// Converts a single-precision OP to BFloat16 value with using rounding mode of
// Round to Nearest Even when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(16) FPConvertBF(bits(32) op, FPCRTYPE fpcr, FPRounding rounding)

    bits(32) result; // BF16 value in top 16 bits
    boolean altfp = HaveAltFP\(\) && !UsingAArch32\(\) && fpcr.AH == '1';
    boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then rounding = FPRounding\_TIEEVEN; // Use RNE rounding mode

    // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_0NaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr);
        else
            result = FPConvertNaN(op);
        if fptype == FPTYPE\_SNaN then
            if fpexc then FPProcessException(FPEXC\_InvalidOp, fpcr);
    elseif fptype == FPTYPE\_Infinity then
        result = FPInfinity(sign);
    elseif fptype == FPTYPE\_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCVBF(value, fpcr, rounding, fpexc);

    // Returns correctly rounded BF16 value from top 16 bits
    return result<31:16>;

// FPConvertBF()
// =====
// Converts a single-precision operand to BFloat16 value.

bits(16) FPConvertBF(bits(32) op, FPCRTYPE fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

Library pseudocode for shared/functions/float/bfloat/FPRoundCVBF

```
// FPRoundCVBF()
// =====
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

bits(32) FPRoundCVBF(real op, FPCRTYPE fpcr, FPRounding rounding, boolean fpexc)
    boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);
```


Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// =====

// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRTYPE fpcr, FPRounding rounding)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding\_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0');
    else
        result = FPRound(real_operand, fpcr, rounding);

    return result;
```

Library pseudocode for shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =====

bits(N) FPAbs(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRTYPE fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPTYPE\_SNaN, FPTYPE\_QNaN} then
                return op; // When fpcr.AH=1, sign of NaN has no consequence

    return '0' : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE;          // Generate floating-point exceptions
    return FPAdd(op1, op2, fpcr, fpexc);

// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    boolean altfmaxfmin = FALSE;    // Do not use altfp mode for FMIN, FMAX and variants
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, fpexc);
    if !done then
        inf1 = (type1 == FPType\_Infinity);  inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);      zero2 = (type2 == FPType\_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcmpare/FPCompare

```
// FPCompare()
// =====

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUntpack(op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op2, fpcr);

    if type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN} then
        result = '0011';
        if type1 == FPTType\_SNaN || type2 == FPTType\_SNaN || signal_nans then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUntpack()
        if value1 == value2 then
            result = '0110';
        elseif value1 < value2 then
            result = '1000';
        else // value1 > value2
            result = '0010';

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcmpareeq/FPCompareEQ

```
// FPCompareEQ()
// =====

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUntpack(op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op2, fpcr);

    if type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN} then
        result = FALSE;
        if type1 == FPTType\_SNaN || type2 == FPTType\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUntpack()
        result = (value1 == value2);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

```
// FPCompareGE()
// =====

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN} then
        result = FALSE;
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =====

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN} then
        result = FALSE;
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// =====

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRTType fpcr, FPRounding rounding)

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr);
        else
            result = FPConvertNaN(op);
            if fptype == FPTType\_SNaN || alt_hp then
                FPProcessException(FPExc\_InvalidOp,fpcr);
    elsif fptype == FPTType\_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
    elsif fptype == FPTType\_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCV(value, fpcr, rounding);

        FPProcessDenorm(fptype, N, fpcr);

    return result;

// FPConvert()
// =====

bits(M) FPConvert(bits(N) op, FPCRTType fpcr)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// =====
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```

Library pseudocode for shared/functions/float/fpcrtype/FPCRTType

```
type FPCRTType;
```

Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// =====
// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)

    case rm of
        when '00' result = FPRounding_TIEAWAY; // A
        when '01' result = FPRounding_TIEEVEN; // N
        when '10' result = FPRounding_POSINF; // P
        when '11' result = FPRounding_NEGINF; // M

    return result;
```

Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// =====
// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)

    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF; // P
        when '10' return FPRounding_NEGINF; // M
        when '11' return FPRounding_ZERO; // Z
```

Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// =====

bits(N) FPDefaultNaN()
    FPCRTType fpcr = FPCR[];
    return FPDefaultNaN(fpcr);

bits(N) FPDefaultNaN(FPCRTType fpcr)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bit sign = if HaveAltFP() && !UsingAArch32() then fpcr.AH else '0';

    bits(E) exp = Ones(E);
    bits(F) frac = '1':Zeros(F-1);

    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =====

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUunpack(op1, fpcr);
    (type2,sign2,value2) = FPUunpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

    if !done then
        inf1 = type1 == FPType_Infinity;
        inf2 = type2 == FPType_Infinity;
        zero1 = type1 == FPType_Zero;
        zero2 = type2 == FPType_Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(fpcr);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);

        if !zero2 then
            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpexc/FPExc

```
enumeration FPExc    {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                      FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

Library pseudocode for shared/functions/float/fpinfinity/FPIInfinity

```
// FPIInfinity()
// =====

bits(N) FPIInfinity(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bits(E) exp = Ones(E);
    bits(F) frac = Zeros(F);

    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmatmul/FPMatMulAdd

```
// FPMatMulAdd()
// =====
//
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, integer esize, FPCRTType fpcr)

    assert N == esize * 2 * 2;
    bits(N) result;
    bits(esize) prod0, prod1, sum;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, esize];
            prod0 = FPMul(Elem[op1, 2*i + 0, esize],
                        Elem[op2, 2*j + 0, esize], fpcr);
            prod1 = FPMul(Elem[op1, 2*i + 1, esize],
                        Elem[op2, 2*j + 1, esize], fpcr);
            sum = FPAAdd(sum, FPAAdd(prod0, prod1, fpcr), fpcr);
            Elem[result, 2*i + j, esize] = sum;

    return result;
```


Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// =====

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMax(op1, op2, fpcr, altfp);

// FPMax()
// =====
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behaviour.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean altfp)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if (altfp && type1 == FPTYPE\_Zero && type2 == FPTYPE\_Zero &&
        ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
        return FPZero(sign2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);

    if !done then
        if value1 > value2 then
            (ftype,sign,value) = (type1,sign1,value1);
        else
            (ftype,sign,value) = (type2,sign2,value2);
        if ftype == FPTYPE\_Infinity then
            result = FPInfinity(sign);
        elsif ftype == FPTYPE\_Zero then
            sign = sign1 AND sign2;          // Use most positive sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

                result = FPRound(value, fpcr, rounding, TRUE);

            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =====

bits(N) FPMaxNormal(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Ones(E-1):'0';
    frac = Ones(F);

    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// =====

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPTType\_QNaN, FPTType\_SNaN};
    boolean type2_nan = type2 IN {FPTType\_QNaN, FPTType\_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPTType\_QNaN && type2 != FPTType\_QNaN then
            op1 = FPInfinity('1');
        elsif type1 != FPTType\_QNaN && type2 == FPTType\_QNaN then
            op2 = FPInfinity('1');

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMax(op1, op2, fpcr, altfmaxfmin);

    return result;
```

Library pseudocode for shared/functions/float/fpmerge/IsMerging

```
// IsMerging()
// =====
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCRTType fpcr)
    boolean merge = HaveAltFP() && !UsingAArch32() && fpcr.NEP == '1';
    return merge;
```

Library pseudocode for shared/functions/float/fpmin/FPMin

```
// FPMin()
// =====

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMIn(op1, op2, fpcr, altfp);

// FPMin()
// =====
// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative behaviour.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean altfp)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if (altfp && type1 == FPTYPE\_Zero && type2 == FPTYPE\_Zero &&
        ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
        return FPZero(sign2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);

    if !done then
        if value1 < value2 then
            (ftype,sign,value) = (type1,sign1,value1);
        else
            (ftype,sign,value) = (type2,sign2,value2);
        if ftype == FPTYPE\_Infinity then
            result = FPInfinity(sign);
        elsif ftype == FPTYPE\_Zero then
            sign = sign1 OR sign2;           // Use most negative sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, TRUE);

    FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// =====

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPTType\_QNaN, FPTType\_SNaN};
    boolean type2_nan = type2 IN {FPTType\_QNaN, FPTType\_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPTType\_QNaN && type2 != FPTType\_QNaN then
            op1 = FPInfinity('0');
        elsif type1 != FPTType\_QNaN && type2 == FPTType\_QNaN then
            op2 = FPInfinity('0');

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMIn(op1, op2, fpcr, altfmaxfmin);

    return result;
```

Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// =====

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTType\_Infinity);
        inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);
        zero2 = (type2 == FPTType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);

            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```



```

// FPMulAdd()
// =====

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// =====
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
    FPCRTType fpcr, boolean fpexc)

    assert N IN {16,32,64};

    (typeA,signA,valueA) = FPUunpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUunpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUunpack(op2, fpcr, fpexc);
    rounding = FPRoundingMode(fpcr);
    inf1 = (type1 == FPTYPE\_Infinity); zero1 = (type1 == FPTYPE\_Zero);
    inf2 = (type2 == FPTYPE\_Infinity); zero2 = (type2 == FPTYPE\_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(HaveAltFP() && UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPTYPE\_0NaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPTYPE\_Infinity); zeroA = (typeA == FPTYPE\_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
            // Other cases involving infinities produce an infinity of the same sign.
            elsif (infA && signA == '0') || (infP && signP == '0') then
                result = FPIfinity('0');
            elsif (infA && signA == '1') || (infP && signP == '1') then
                result = FPIfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);

    if !invalidop && fpexc then

```

```
FPProcessDenorms3(typeA, type1, type2, N, fpcr);  
return result;
```



```

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
    boolean fpxc = TRUE; // Generate floating-point exceptions
    return
    assert N == 32;
    rounding = FPMulAddH(addend, op1, op2, fpcr, fpxc);

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
    FPCRTYPE fpcr, boolean fpxc)

    assert N == 32;
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUntpack(addend, fpcr, fpxc);
    (addend, fpcr);
    (type1,sign1,value1) = FPUntpack(op1, fpcr, fpxc);
    (op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op2, fpcr, fpxc);
    (op2, fpcr);
    inf1 = (type1 == FPTYPE_Infinity); zero1 = (type1 == FPTYPE_Zero);
    inf2 = (type2 == FPTYPE_Infinity); zero2 = (type2 == FPTYPE_Zero);

    (done,result) = FPPROCESSNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpxc);
);
    (done,result) = FPPROCESSNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPTYPE_0NaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr);
            if fpxc then(fpcr); FPPROCESSException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPTYPE_Infinity); zeroA = (typeA == FPTYPE_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr);
            if fpxc then(fpcr); FPPROCESSException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';

```

```

        result = FPZero(result_sign);
    else
        result = FPRound(result_value, fpcr, rounding, fpexc);
    (result_value, fpcr);

    if !invalidop && fpexc then if !invalidop then
        FPProcessDenorm(typeA, N, fpcr);

    return result;

```

Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

```

// FPProcessNaNs3H()
// =====

(boolean, bits(N)) (boolean, bits(N)) FPProcessNaNs3H( FPType type1, FPType type2, FPType
bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
FPCRTYPE fpcr, boolean fpexc)
fpcr)

assert N IN {32,64};

bits(N) result;
// When TRUE, use alternative NaN propagation rules.
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean op1_nan = type1 IN {FPTYPE_SNaN, FPTYPE_QNaN};
boolean op2_nan = type2 IN {FPTYPE_SNaN, FPTYPE_QNaN};
boolean op3_nan = type3 IN {FPTYPE_SNaN, FPTYPE_QNaN};
boolean fpexc = TRUE;
if altfp then
    if (type1 == FPTYPE_SNaN || type2 == FPTYPE_SNaN || type3 == FPTYPE_SNaN) then
        type_nan = FPTYPE_SNaN;
    else
        type_nan = FPTYPE_QNaN;

    if altfp && op1_nan && op2_nan && op3_nan then // <n> register NaN selected
    if altfp && op1_nan && op2_nan && op3_nan then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc));
    elsif altfp && op2_nan && (op1_nan || op3_nan) then // <n> register NaN selected
    (type_nan, op2, fpcr, fpexc)); // <n> register NaN selected
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc));
    elsif altfp && op3_nan && op1_nan then // <m> register NaN selected
    (type_nan, op2, fpcr, fpexc)); // <n> register NaN selected
    elsif altfp && op3_nan && op1_nan then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op3, fpcr, fpexc));
    (type_nan, op3, fpcr, fpexc)); // <m> register NaN selected
    elsif type1 == FPTYPE_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPTYPE_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc));
    elsif type3 == FPTYPE_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc));
    elsif type1 == FPTYPE_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPTYPE_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc));
    elsif type3 == FPTYPE_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc));
    else
        done = FALSE; result = Zeros(); // 'Don't care' result
    return (done, result);

```

Library pseudocode for shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// =====

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    bits(N) result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTType\_Infinity);
        inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);
        zero2 = (type2 == FPTType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2);
        elsif inf1 || inf2 then
            result = FPIInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);

            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =====

bits(N) FPNeg(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRTType fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPTType\_SNaN, FPType\_QNaN} then

                return op;          // When fpcr.AH=1, sign of NaN has no consequence

    return NOT(op<N-1>) : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// =====

bits(N) FPOnePointFive(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPPProcessDenorm

```
// FPPProcessDenorm()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPPProcessDenorm(FPType fptype, integer N, FPCRTYPE fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPType_Denormal then
        FPPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPPProcessDenorms

```
// FPPProcessDenorms()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPPProcessDenorms(FPType type1, FPType type2, integer N, FPCRTYPE fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal) then
        FPPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPPProcessDenorms3

```
// FPPProcessDenorms3()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPPProcessDenorms3(FPType type1, FPType type2, FPType type3, integer N, FPCRTYPE fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal) then
        FPPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPPProcessDenorms4

```
// FPPProcessDenorms4()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPPProcessDenorms4(FPType type1, FPType type2, FPType type3, FPType type4, integer N, FPCRTYPE fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal || type4 == FPType_Denormal) then
        FPPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// =====
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRTYPE fpcr)

// Determine the cumulative exception bit number
case exception of
    when FPExc_InvalidOp      cumul = 0;
    when FPExc_DivideByZero    cumul = 1;
    when FPExc_Overflow        cumul = 2;
    when FPExc_Underflow       cumul = 3;
    when FPExc_Inexact         cumul = 4;
    when FPExc_InputDenorm     cumul = 7;
enable = cumul + 8;
if fpcr<enable> == '1' then
    // Trapping of the exception enabled.
    // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
    // if so then how exceptions may be accumulated before calling FPTrappedException()
    IMPLEMENTATION_DEFINED "floating-point trap handling";
elsif UsingAArch32() then
    // Set the cumulative exception bit
    FPSCR<cumul> = '1';
else
    // Set the cumulative exception bit
    FPSR<cumul> = '1';

return;
```

Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// =====
//
bits(N) FPProcessNaN(FPTYPE fptype, bits(N) op, FPCRTYPE fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPProcessNaN(fptype, op, fpcr, fpexc);

// FPProcessNaN()
// =====
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPProcessNaN(FPTYPE fptype, bits(N) op, FPCRTYPE fpcr, boolean fpexc)

    assert N IN {16,32,64};
    assert fptype IN {FPTYPE_ONaN, FPTYPE_SNaN};

    case N of
        when 16 topfrac = 9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPTYPE_SNaN then
        result<topfrac> = '1';
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then // DefaultNaN requested
        result = FPDefaultNaN(fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// =====

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1,
                                bits(N) op2, FPCRType fpcr)
    boolean altfmaxfmin = FALSE;    // Do not use alfp mode for FMIN, FMAX and variants
    boolean fpexc       = TRUE;     // Generate floating-point exceptions
    return FPProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, fpexc);

// FPProcessNaNs()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behaviour for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2,
                                FPCRType fpcr, boolean altfmaxfmin, boolean fpexc)

    assert N IN {16,32,64};
    boolean altfp      = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan    = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan    = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean any_snan    = type1 == FPType_SNaN || type2 == FPType_SNaN;
    FPType type_nan     = if any_snan then FPType_SNaN else FPType_QNaN;

    if altfmaxfmin && (op1_nan || op2_nan) then
        FPProcessException(FPExc_InvalidOp, fpcr);
        done = TRUE; sign2 = op2<N-1>;
        result = if type2 == FPType_Zero then FPZero(sign2) else op2;
    elsif altfp && op1_nan && op2_nan then
        done = TRUE; result = FPProcessNaN(type_nan, op1, fpcr, fpexc);    // <n> register NaN selected
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
    else
        done = FALSE; result = Zeros();    // 'Don't care' result

    return (done, result);
```

Library pseudocode for shared/functions/float/fpprocessnans3/FPPProcessNaNs3

```
// FPPProcessNaNs3()
// =====

(boolean, bits(N)) FPPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N) op2, bits(N) op3,
                                   FPCRTType fpcr)
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPPProcessNaNs3()
// =====
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FPPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N) op2, bits(N) op3,
                                   FPCRTType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};

    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp then
        if type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;

    if altfp && op1_nan && op2_nan && op3_nan then
        done = TRUE; result = FPPProcessNaN(type_nan, op2, fpcr, fpexc);    // <n> register NaN selected
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
        done = TRUE; result = FPPProcessNaN(type_nan, op2, fpcr, fpexc);    // <n> register NaN selected
    elsif altfp && op3_nan && op1_nan then
        done = TRUE; result = FPPProcessNaN(type_nan, op3, fpcr, fpexc);    // <m> register NaN selected
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPPProcessNaN(type3, op3, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPPProcessNaN(type3, op3, fpcr, fpexc);
    else
        done = FALSE; result = Zeros();    // 'Don't care' result

    return (done, result);
```



```

// FPRecipEstimate()
// =====

bits(N) FPRecipEstimate(bits(N) operand, FPCRTType fpcr)

    assert N IN {16,32,64};

    // When using alternative floating-point behaviour, do not generate
    // floating-point exceptions, flush denormal input and output to zero,
    // and use RNE rounding mode.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';
    if altfp then fpcr.RMode = '00';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    FPRounding rounding = FPRoundingMode(fpcr);
    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPTType\_Infinity then
        result = FPZero(sign);
    elsif fptype == FPTType\_Zero then
        result = FPInfinity(sign);
        if fpexc then FPProcessException(FPExc\_DivideByZero, fpcr);
    elsif (
        (N == 16 && Abs(value) < 2.0^-16) ||
        (N == 32 && Abs(value) < 2.0^-128) ||
        (N == 64 && Abs(value) < 2.0^-1024)
    ) then
        case rounding of
            when FPRounding\_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding\_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding\_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding\_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign) else FPMMaxNormal(sign);
    if fpexc then
        FPProcessException(FPExc\_Overflow, fpcr);
        FPProcessException(FPExc\_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
        && (
            (N == 16 && Abs(value) >= 2.0^14) ||
            (N == 32 && Abs(value) >= 2.0^126) ||
            (N == 64 && Abs(value) >= 2.0^1022)
        ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign);

        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64

```

```

        fraction = operand<51:0>;
        exp = UInt(operand<62:52>);

if exp == 0 then
    if fraction<51> == '0' then
        exp = -1;
        fraction = fraction<49:0>:'00';
    else
        fraction = fraction<50:0>:'0';

integer scaled;
boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

if !increasedprecision then
    scaled = UInt('1':fraction<51:44>);
else
    scaled = UInt('1':fraction<51:41>);

case N of
    when 16 result_exp = 29 - exp; // In range 29-30 = -1 to 29+1 = 30
    when 32 result_exp = 253 - exp; // In range 253-254 = -1 to 253+1 = 254
    when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

// Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
// fixed-point number in range [0.5 .. 1.0].
estimate = RecipEstimate(scaled, increasedprecision);

// Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
// fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit,
// high-order bits from estimate, and exponent calculated above.
if !increasedprecision then
    fraction = estimate<7:0> : Zeros(44);
else
    fraction = estimate<11:0> : Zeros(40);

if result_exp == 0 then
    fraction = '1' : fraction<51:1>;
elsif result_exp == -1 then
    fraction = '01' : fraction<51:2>;
    result_exp = 0;

case N of
    when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
    when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
    when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

return result;

```

Library pseudocode for shared/functions/float/fpreciestimate/RecipEstimate

```
// RecipEstimate()
// =====
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range  $0.5 \leq x < 1.0$ .
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

integer RecipEstimate(integer a, boolean increasedprecision)

    integer r;
    if !increasedprecision then
        assert 256 <= a && a < 512;
        a = a*2+1; // Round to nearest
        integer b = (2 ^ 19) DIV a;
        r = (b+1) DIV 2; // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;
        a = a*2+1; // Round to nearest
        real real_val = Real(2^25)/Real(a);
        r = RoundDown(real_val);
        real error = real_val - Real(r);
        boolean round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
        if round_up then r = r+1;
        assert 4096 <= r && r < 8192;

    return r;
```

Library pseudocode for shared/functions/float/fprecpX/FPRecpX

```
// FPRecpX()
// =====

bits(N) FPRecpX(bits(N) op, FPCRTType fpcr)

    assert N IN {16,32,64};

    case N of
        when 16 esize = 5;
        when 32 esize = 8;
        when 64 esize = 11;

    bits(N)          result;
    bits(esize)      exp;
    bits(esize)      max_exp;
    bits(N-(esize+1)) frac = Zeros();

    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;           // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    (fptype,sign,value) = FPUntpack(op, fpcr, fpexc);

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp = Ones(esize) - 1;

    if fptype == FType\_SNaN || fptype == FType\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then           // Zero and denormals
            result = sign:max_exp:frac;
        else                          // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;
```

Library pseudocode for shared/functions/float/fpround/FPRound

```
// FPRound()
// =====
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real op, FPCRTType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =====
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRound(real op, FPCRTType fpcr, FPRounding rounding, boolean fpexc)
    fpcr.AHP = '0';
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =====

bits(N) FPRound(real op, FPCRTType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
```



```

// FPRoundBase()
// =====

bits(N) FPRoundBase(real op, FPCRTType fpcr, FPRounding rounding, boolean isbfloat16)
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRoundBase()
// =====
// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRTType fpcr, FPRounding rounding,
    boolean isbfloat16, boolean fpexc)

    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding\_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14; E = 5; F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126; E = 8; F = 7;
    elsif N == 32 then
        minimum_exp = -126; E = 8; F = 23;
    else // N == 64
        minimum_exp = -1022; E = 11; F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0; exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0; exponent = exponent + 1;

    // When TRUE, detection of underflow occurs after rounding and the test for a
    // denormalized number for single and double precision values occurs after rounding.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    // Deal with flush-to-zero before rounding if FPCR.AH != '1'.
    if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
        exponent < minimum_exp) then
        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
            FPSR.UFC = '1';
    return FPZero(sign);

    biased_exp_unconstrained = exponent - minimum_exp + 1;
    int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
    error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.

```

```

int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
if !altfp && biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
    if fpexc then FPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
if altfp then
    case rounding of
        when FPRounding_TIEEVEN
            round_up_unconstrained = (error_unconstrained > 0.5 ||
                (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up_unconstrained = FALSE;
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up_unconstrained then
        int_mant_unconstrained = int_mant_unconstrained + 1;
        if int_mant_unconstrained == 2^(F+1) then // Rounded up to next exponent
            biased_exp_unconstrained = biased_exp_unconstrained + 1;
            int_mant_unconstrained = int_mant_unconstrained DIV 2;

// Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
    // the result of unconstrained rounding is less than the minimum normalized number
    if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then // Flush-to-zero
        if fpexc then
            FPSR.UFC = '1';
            FPProcessException(FPExc_Inexact, fpcr);
        return FPZero(sign);
    elseif error != 0.0 || fpcr.UFE == '1' then
        if fpexc then FPProcessException(FPExc_Underflow, fpcr);
else // altfp == FALSE
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then // Rounded up from denormalized to normalized
            biased_exp = 1;
        if int_mant == 2^(F+1) then // Rounded up to next exponent
            biased_exp = biased_exp + 1;
            int_mant = int_mant DIV 2;

// Handle rounding to odd
if error != 0.0 && rounding == FPRounding_ODD then

```



```

    int_mant<0> = '1';

// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then
        result = if overflow_to_inf then FPInfinity(sign) else FPMMaxNormal(sign);
        if fpexc then FPPProcessException(FPExc\_Overflow, fpcr);
        error = 1.0; // Ensure that an Inexact exception occurs
    else
        result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
else // Alternative half precision
    if biased_exp >= 2^E then
        result = sign : Ones(N-1);
        if fpexc then FPPProcessException(FPExc\_InvalidOp, fpcr);
        error = 0.0; // Ensure that an Inexact exception does not occur
    else
        result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

// Deal with Inexact exception.
if error != 0.0 then
    if fpexc then FPPProcessException(FPExc\_Inexact, fpcr);

return result;

```

Library pseudocode for shared/functions/float/fpround/FPRoundCV

```

// FPRoundCV()
// =====
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRTType fpcr, FPRounding rounding)
    fpcr.FZ16 = '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

```

Library pseudocode for shared/functions/float/fprounding/FPRounding

```

enumeration FPRounding {FPRounding_TIEEVEN, FPRounding_POSINF,
                        FPRounding_NEGINF, FPRounding_ZERO,
                        FPRounding_TIEAWAY, FPRounding_ODD};

```

Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

```

// FPRoundingMode()
// =====
// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRTType fpcr)
    return FPDecodeRounding(fpcr.RMode);

```

Library pseudocode for shared/functions/float/fproundint/FPRoundInt

```
// FPRoundInt()
// =====

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCRTYPE fpcr, FPRounding rounding, boolean exact)

    assert rounding != FPRounding\_ODD;
    assert N IN {16,32,64};

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUntpack(op, fpcr, fpexc);

    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elseif fptype == FPTYPE\_Infinity then
        result = FPInfinity(sign);
    elseif fptype == FPTYPE\_Zero then
        result = FPZero(sign);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        case rounding of
            when FPRounding\_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding\_POSINF
                round_up = (error != 0.0);
            when FPRounding\_NEGINF
                round_up = FALSE;
            when FPRounding\_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding\_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value.
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = FPZero(sign);
        else
            result = FPRound(real_result, fpcr, FPRounding\_ZERO);

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc\_Inexact, fpcr);

    return result;
```



```

// FPRoundIntN()
// =====

bits(N) FPRoundIntN(bits(N) op, FPCRTType fpcr, FPRounding rounding, integer intsize)
    assert rounding != FPRounding\_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP\(\) && !UsingAArch32\(\) && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype IN {FPTType\_SNaN, FPTType\_0NaN, FPTType\_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
            FPProcessException(FPExc\_InvalidOp, fpcr);
    elseif fptype == FPTType\_Zero then
        result = FPZero(sign);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        case rounding of
            when FPRounding\_TIEEVEN
                round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
            when FPRounding\_POSINF
                round_up = error != 0.0;
            when FPRounding\_NEGINF
                round_up = FALSE;
            when FPRounding\_ZERO
                round_up = error != 0.0 && int_result < 0;
            when FPRounding\_TIEAWAY
                round_up = error > 0.5 || (error == 0.5 && int_result >= 0);

        if round_up then int_result = int_result + 1;
        overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);

        if overflow then
            if N == 32 then
                exp = 126 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            else
                exp = 1022 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
                FPProcessException(FPExc\_InvalidOp, fpcr);
                // This case shouldn't set Inexact.
                error = 0.0;

        else
            // Convert integer value into an equivalent real value.
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact.
            if real_result == 0.0 then
                result = FPZero(sign);
            else
                result = FPRound(real_result, fpcr, FPRounding\_ZERO);

```

```
// Generate inexact exceptions.  
if error != 0.0 then  
    FPProcessException\(FPExc\_Inexact, fpcr\);  
return result;
```



```

// FPRsqrtEstimate()
// =====

bits(N) FPRsqrtEstimate(bits(N) operand, FPCRTType fpcr)

    assert N IN {16,32,64};

    // When using alternative floating-point behaviour, do not generate
    // floating-point exceptions and flush denormal input to zero.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    if fptype == FPTType\_SNaN || fptype == FPTType\_0NaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elseif fptype == FPTType\_Zero then
        result = FPInfinity(sign);
        if fpexc then FPProcessException(FPExc\_DivideByZero, fpcr);
    elseif sign == '1' then
        result = FPDefaultNaN(fpcr);
        if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
    elseif fptype == FPTType\_Infinity then
        result = FPZero('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == '0' do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        integer scaled;
        boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

        if !increasedprecision then
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:44>);
            else
                scaled = UInt('01':fraction<51:45>);
        else
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:41>);
            else
                scaled = UInt('01':fraction<51:42>);

        case N of
            when 16 result_exp = ( 44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled, increasedprecision);

        // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a

```

```

// fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit and high-order
// fraction bits, and exponent calculated above.
case N of
  when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros(2);
  when 32
    if !increasedprecision then
      result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
    else
      result = '0' : result_exp<N-25:0> : estimate<11:0>:Zeros(11);
  when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);

return result;

```


Library pseudocode for shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

```
// RecipSqrtEstimate()
// =====
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range  $0.25 \leq x < 1.0$ .
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

integer RecipSqrtEstimate(integer a, boolean increasedprecision)

    integer r;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then // 0.25 .. 0.5
            a = a*2+1; // a in units of 1/512 rounded to nearest
        else // 0.5 .. 1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = (a+1)*2; // a in units of 1/256 rounded to nearest
        integer b = 512;
        while a*(b+1)*(b+1) < 2^28 do
            b = b+1;
        // b = largest b such that  $b < 2^{14} / \sqrt{a}$ 
        r = (b+1) DIV 2; // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        real real_val;
        real error;
        integer int_val;

        if a < 2048 then // 0.25... 0.5
            a = a*2 + 1; // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a)/2.0;
        else // 0.5..1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = a+1; // Taking 10 bits of the fraction and force a 1 at the bottom
            real_val = Real(a);

        real_val = Sqrt(real_val); // This number will lie in the range of 32 to 64
        // Round to nearest even for a DP float number
        real_val = real_val * Real(2^47); // The integer is the size of the whole DP mantissa
        int_val = RoundDown(real_val); // Calculate rounding value
        error = real_val - Real(int_val);
        round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
        if round_up then int_val = int_val+1;

        real_val = Real(2^65)/Real(int_val); // Lies in the range  $4096 \leq \text{real\_val} < 8192$ 
        int_val = RoundDown(real_val); // Round that (to nearest even) to give integer
        error = real_val - Real(int_val);
        round_up = (error > 0.5 || (error == 0.5 && int_val<0> == '1'));
        if round_up then int_val = int_val+1;

        r = int_val;
        assert 4096 <= r && r < 8192;

    return r;
```

Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// =====

bits(N) FPSqrt(bits(N) op, FPCRTType fpcr)

    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPTType\_SNaN || fptype == FPTType\_0NaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elseif fptype == FPTType\_Zero then
        result = FPZero(sign);
    elseif fptype == FPTType\_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elseif sign == '1' then
        result = FPDefaultNaN(fpcr);
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr);

        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpsub/FPSub

```
// FPSub()
// =====

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTType\_Infinity);
        inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);
        zero2 = (type2 == FPTType\_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elseif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);

            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpthree/FPThree

```
// FPThree()
// =====

bits(N) FPThree(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// =====

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRTYPE fpcr, FPRounding rounding)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding\_ODD;

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP\(\) && !UsingAArch32\(\) && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    // If NaN, set cumulative flag or take exception.
    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_0NaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    case rounding of
        when FPRounding\_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding\_POSINF
            round_up = (error != 0.0);
        when FPRounding\_NEGINF
            round_up = FALSE;
        when FPRounding\_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding\_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions.
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc\_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

```
// FPToFixedJS()
// =====

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCRTType fpcr, boolean Is64)

    assert M == 64 && N == 32;

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpxc_idenorm = !(HaveAltFP() && UsingAArch32() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUntpack(op, fpcr, fpxc_idenorm);

    Z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);
        Z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc\_InvalidOp, fpcr);
        Z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc\_Inexact, fpcr);
        Z = '0';
    elsif sign == '1' && value == 0.0 then
        Z = '0';
    elsif sign == '0' && value == 0.0 && IsZero(op<51:0>) then
        Z = '0';

    if fptype == FPTType\_Infinity then result = 0;

    return (result<N-1:0>, Z);
```

Library pseudocode for shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =====

bits(N) FPTwo(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/fptype/FPType

```
enumeration FPType {FPType_Zero,  
                    FPType_Denormal,  
                    FPType_Nonzero,  
                    FPType_Infinity,  
                    FPType_QNaN,  
                    FPType_SNaN};
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()  
// =====  
  
(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRTType fpcr)  
    fpcr.AHP = '0';  
    boolean fpexc = TRUE;    // Generate floating-point exceptions  
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);  
    return (fp_type, sign, value);  
  
// FPUnpack()  
// =====  
//  
// Used by data processing and int/fixed <-> FP conversion instructions.  
// For half-precision data it ignores AHP, and observes FZ16.  
  
(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRTType fpcr, boolean fpexc)  
    fpcr.AHP = '0';  
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);  
    return (fp_type, sign, value);
```



```

// FPUunpackBase()
// =====

(FPType, bit, real) FPUunpackBase(bits(N) fpval, FPCRType fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    (fp_type, sign, value) = FPUunpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

// FPUunpackBase()
// =====
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(FPType, bit, real) FPUunpackBase(bits(N) fpval, FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};

    boolean altfp = HaveAltFP() && !UsingAArch32();
    boolean fiz   = altfp && fpcr.FIZ == '1';
    boolean fz     = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');

    if N == 16 then
        sign    = fpval<15>;
        exp16   = fpval<14:10>;
        frac16  = fpval<9:0>;
        if IsZero(exp16) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPType\_Zero; value = 0.0;
            else
                fptype = FPType\_Denormal; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FPType\_Infinity; value = 2.0^1000000;
            else
                fptype = if frac16<9> == '1' then FPType\_QNaN else FPType\_SNaN;
                value = 0.0;
        else
            fptype = FPType\_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 then
        sign    = fpval<31>;
        exp32   = fpval<30:23>;
        frac32  = fpval<22:0>;

        if IsZero(exp32) then
            if IsZero(frac32) then
                // Produce zero if value is zero.
                fptype = FPType\_Zero; value = 0.0;
            elsif fiz || fiz then // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FPType\_Zero; value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPPProcessException(FPExc\_InputDenorm, fpcr);
            else
                fptype = FPType\_Denormal; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                fptype = FPType\_Infinity; value = 2.0^1000000;
            else

```



```

        fptype = if frac32<22> == '1' then FPTYPE\_QNaN else FPTYPE\_SNaN;
        value = 0.0;
    else
        fptype = FPTYPE\_Nonzero;
        value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);
else // N == 64
    sign = fpval<63>;
    exp64 = fpval<62:52>;
    frac64 = fpval<51:0>;

    if IsZero(exp64) then
        if IsZero(frac64) then
            // Produce zero if value is zero.
            fptype = FPTYPE\_Zero; value = 0.0;
        elsif fz || fiz then // Flush-to-zero if FIZ==1 or AH,FZ==01
            fptype = FPTYPE\_Zero; value = 0.0;
            // Check whether to raise Input Denormal floating-point exception.
            // fpcr.FIZ==1 does not raise Input Denormal exception.
            if fz then
                // Denormalized input flushed to zero
                if fpexc then FPPProcessException(FPExc\_InputDenorm, fpcr);
            else
                fptype = FPTYPE\_Denormal; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                fptype = FPTYPE\_Infinity; value = 2.0^1000000;
            else
                fptype = if frac64<51> == '1' then FPTYPE\_QNaN else FPTYPE\_SNaN;
                value = 0.0;
        else
            fptype = FPTYPE\_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;

    return (fptype, sign, value);

```

Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

```

// FPUnpackCV()
// =====
//
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRTYPE fpcr)
    fpcr.FZ16 = '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

```

Library pseudocode for shared/functions/float/fpzero/FPZero

```

// FPZero()
// =====

bits(N) FPZero(bit sign)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Zeros(E);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;

```

Library pseudocode for shared/functions/float/vfpexpandimm/VFPExpandImm

```
// VFPExpandImm()
// =====

bits(N) VFPExpandImm(bits(8) imm8)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

Library pseudocode for shared/functions/memory/AArch64.BranchAddr

```
// AArch64.BranchAddr()
// =====
// Return the virtual address with tag bits removed for storing to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, PSTATE.EL);
    if msbit == 63 then
        return vaddress;
    elsif (PSTATE.EL IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>);
    else
        return ZeroExtend(vaddress<msbit:0>);
```

Library pseudocode for shared/functions/memory/AccType

```
enumeration AccType {AccType_NORMAL, AccType_VEC,           // Normal loads and stores
                     AccType_STREAM, AccType_VECSTREAM,     // Streaming loads and stores
                     AccType_A32LSMD,                      // Load and store multiple
                     AccType_ATOMIC, AccType_ATOMICRW,      // Atomic loads and stores
                     AccType_ORDERED, AccType_ORDEREDRW,    // Load-Acquire and Store-Release
                     AccType_ORDEREDATOMIC,                 // Load-Acquire and Store-Release with atomic access
                     AccType_ORDEREDATOMICRW,
                     AccType_ATOMICLS64,                    // Atomic 64-byte loads and stores
                     AccType_LIMITEDORDERED,                // Load-LOAcquire and Store-LORelease
                     AccType_UNPRIV,                        // Load and store unprivileged
                     AccType_IFETCH,                        // Instruction fetch
                     AccType_TTW,                           // Translation table walk
                     AccType_NONFAULT,                      // Non-faulting loads
                     AccType_CNOTFIRST,                     // Contiguous FF load, not first element
                     AccType_NV2REGISTER,                   // MRS/MSR instruction used at EL1 and which is
                                                             // to a memory access that uses the EL2 translation
                                                             // Other operations
                     AccType_DC,                            // Data cache maintenance
                     AccType_IC,                            // Instruction cache maintenance
                     AccType_DCZVA,                        // DC ZVA instructions
                     AccType_ATPAN,                        // Address translation with PAN permission check
                     AccType_AT};                          // Address translation
```

Library pseudocode for shared/functions/memory/AccessDescriptor

```
type AccessDescriptor is (
    AccType acctype,
    MPAMInfo mpam,
    AccType acctype)mpam,
    boolean page_table_walk,
    boolean secondstage,
    boolean s2fslwalk,
    integer level
)
```

Library pseudocode for shared/functions/memory/AddrTop

```
// AddrTop()
// =====
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;
```

Library pseudocode for shared/functions/memory/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord fault, // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress address,
    bits(64) vaddress
)
```

Library pseudocode for shared/functions/memory/Allocation

```
constant bits(2) MemHint_No = '00';    // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01';    // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10';    // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';   // Read-Allocate, Write-Allocate
```

Library pseudocode for shared/functions/memory/BigEndian

```
// BigEndian()
// =====

boolean BigEndian(AccType acctype)
    boolean bigend;
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        return SCTL_EL2.EE == '1';

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[0].EE != '0');
    else
        bigend = (SCTLR[0].EE != '0');
    return bigend;
```

Library pseudocode for shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// =====

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```

Library pseudocode for shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00';    // Non-cacheable
constant bits(2) MemAttr_WT = '10';    // Write-through
constant bits(2) MemAttr_WB = '11';    // Write-back
```

Library pseudocode for shared/functions/memory/CreateAccessDescriptorTTW

```
// CreateAccessDescriptorTTW()
// =====

AccessDescriptor CreateAccessDescriptorTTW(AccType acctype, boolean secondstage,
                                           boolean s2fslwalk, integer level)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype);
    accdesc.page_table_walk = TRUE;
    accdesc.s2fslwalk = s2fslwalk;
    accdesc.secondstage = secondstage;
    accdesc.level = level;
    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccessDescriptor

```
// CreateAccessDescriptor()
// =====

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype);
    accdesc.page_table_walk = FALSE;
    return accdesc;
```

Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean nXS);
```

Library pseudocode for shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

Library pseudocode for shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// =====
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if HavePACExt() then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                if HavePACExt() then
                    tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
            else
                tbi = TCR_EL2.TBI;
                if HavePACExt() then tbid = TCR_EL2.TBID;
        when EL3
            tbi = TCR_EL3.TBI;
            if HavePACExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr) then '1' else '0');
```

Library pseudocode for shared/functions/memory/EffectiveTCMA

```
// EffectiveTCMA()
// =====
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;

    return tcma;
```

Library pseudocode for shared/functions/memory/Fault

```
enumeration Fault {Fault_None,
    Fault_AccessFlag,
    Fault_Alignment,
    Fault_Background,
    Fault_Domain,
    Fault_Permission,
    Fault_Translation,
    Fault_AddressSize,
    Fault_SyncExternal,
    Fault_SyncExternalOnWalk,
    Fault_SyncParity,
    Fault_SyncParityOnWalk,
    Fault_AsyncParity,
    Fault_AsyncExternal,
    Fault_Debug,
    Fault_TLBConflict,
    Fault_BranchTarget,
    Fault_HWUpdateAccessFlag,
    Fault_Lockdown,
    Fault_Exclusive,
    Fault_ICacheMaint};
```

Library pseudocode for shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault    statuscode, // Fault Status
    AccType acctype,        // Type of access that faulted
    FullAddress ipaddress, // Intermediate physical address
    boolean s2fslwalk,     // Is on a Stage 1 translation table walk
    boolean write,         // TRUE for a write, FALSE for a read
    integer level,         // For translation, access flag and permission faults
    bit extflag,           // IMPLEMENTATION DEFINED syndrome for External aborts
    boolean secondstage,   // Is a Stage 2 abort
    bits(4) domain,       // Domain number, AArch32 only
    bits(2) errortype,     // [Armv8.2 RAS] AArch32 AET or AArch64 SET
    bits(4) debugmoe)     // Debug method of entry, from AArch32 only
```

Library pseudocode for shared/functions/memory/FullAddress

```
type FullAddress is (
    PASpace paspace,
    bits(52) address
)
```

Library pseudocode for shared/functions/memory/Hint_Prefetch

```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

Library pseudocode for shared/functions/memory/MBReqDomain

```
enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                           MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

Library pseudocode for shared/functions/memory/MBReqTypes

```
enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

Library pseudocode for shared/functions/memory/MPAM

```
type PARTIDtype = bits(16);
type PMGtype = bits(8);
type PARTIDspaceType = bit;
constant PARTIDspaceType PIdSpace_Secure    = '0';
constant PARTIDspaceType PIdSpace_NonSecure = '1';

type MPAMinfo is (
    PARTIDspaceType mpam_ns,
    PARTIDtype partid,
    PMGtype pmg
)
```

Library pseudocode for shared/functions/memory/MemAttrHints

```
type MemAttrHints is (
    bits(2) attrs, // See MemAttr_*, Cacheability attributes
    bits(2) hints, // See MemHint_*, Allocation hints
    boolean transient
)
```

Library pseudocode for shared/functions/memory/MemType

```
enumeration MemType {MemType_Normal, MemType_Device};
```

Library pseudocode for shared/functions/memory/MemoryAttributes

```
type MemoryAttributes is (
    MemType memtype,
    DeviceType device, // For Device memory types
    MemAttrHints inner, // Inner hints and attributes
    MemAttrHints outer, // Outer hints and attributes
    Shareability shareability, // Shareability attribute
    boolean tagged, // Tagged access
    bit xs // XS attribute
)
```

Library pseudocode for shared/functions/memory/PASpace

```
enumeration PASpace {
    PAS_NonSecure,
    PAS_Secure,
};
```

Library pseudocode for shared/functions/memory/Permissions

```
type Permissions is (
    bits(2) ap_table,    // Stage 1 hierarchical access permissions
    bit     xn_table,    // Stage 1 hierarchical execute-never for single EL regimes
    bit     pxn_table,   // Stage 1 hierarchical privileged execute-never
    bit     uxn_table,   // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap,          // Stage 1 access permissions
    bit     xn,          // Stage 1 execute-never for single EL regimes
    bit     uxn,         // Stage 1 unprivileged execute-never
    bit     pxn,         // Stage 1 privileged execute-never
    bits(2) s2ap,        // Stage 2 access permissions
    bit     s2xn,        // Stage 2 extended execute-never
    bit     s2xn         // Stage 2 execute-never
)
```

Library pseudocode for shared/functions/memory/PhysMemRead

```
// Returns the value read from memory, and a PhysMemRetStatus.
// If there is an external abort on the read, the PhysMemRetStatus indicates this
// and the value is UNKNOWN.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.
(PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor desc, integer size, AccessDescriptor accdesc)
```

Library pseudocode for shared/functions/memory/PhysMemRetStatus

```
type PhysMemRetStatus is (Fault statuscode, // Fault Status
    bit extflag, // IMPLEMENTATION DEFINED
    // syndrome for External aborts
    bits(2) errortype, // optional error state
    // returned on a physical
    // memory access
    bits(64) store64bstatus, // status of 64B store
    AccType acctype) // Type of access that faulted
```

Library pseudocode for shared/functions/memory/PhysMemWrite

```
// Writes the value to memory, and returns a PhysMemRetStatus.
// If there is an external abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.
PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, AccessDescriptor accdesc, bits(8*size) value)
```

Library pseudocode for shared/functions/memory/PrefetchHint

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

Library pseudocode for shared/functions/memory/Shareability

```
enumeration Shareability {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```
SpeculativeStoreBypassBarrierToPA();
```


Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```
SpeculativeStoreBypassBarrierToVA();
```

Library pseudocode for shared/functions/memory/Tag

```
constant integer LOG2_TAG_GRANULE = 4;  
constant integer TAG_GRANULE = 1 << LOG2_TAG_GRANULE;
```

Library pseudocode for shared/functions/memory/_Mem

```
// These two _Mem[] accessors are the hardware operations which perform single-copy atomic,  
// aligned, little-endian memory accesses of size bytes from/to the underlying physical  
// memory array of bytes.  
//  
// The functions address the array using desc.paddress which supplies:  
// * A 52-bit physical address  
// * A single NS bit to select between Secure and Non-secure parts of the array.  
//  
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,  
// etc and other parameters required to access the physical memory or for setting syndrome  
// register in the event of an External abort.  
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc];  
  
_Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc] = bits(8*size) value;
```

Library pseudocode for shared/functions/mpam/DefaultMPAMInfo

```
// DefaultMPAMInfo()  
// =====  
// Returns default MPAM info. The partidspace argument sets  
// the PARTID space of the default MPAM information returned.  
  
MPAMInfo DefaultMPAMInfo(PARTIDspaceType partidspace)  
    MPAMInfo DefaultInfo;  
    DefaultInfo.mpam_ns = partidspace;  
    DefaultInfo.partid = DefaultPARTID;  
    DefaultInfo.pmg = DefaultPMG;  
    return DefaultInfo;
```

Library pseudocode for shared/functions/mpam/DefaultPARTID

```
constant PARTIDtype DefaultPARTID = 0<15:0>;
```

Library pseudocode for shared/functions/mpam/DefaultPMG

```
constant PMGtype DefaultPMG = 0<7:0>;
```

Library pseudocode for shared/functions/mpam/GenMPAMcurEL

```
// GenMPAMcurEL()
// =====
// GenMPAMcurEL
// =====
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in a version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(AccType acctype)
    bits(2) mpamEL;
    boolean validEL = FALSE;
    SecurityState security = if IsSecure() then SS_Secure else SS_NonSecure;
    boolean InD = FALSE;
    PARTIDspaceType pspace = pspace;
    if PARTIDspaceFromSS(security);
    if pspace == PIdSpace_NonSecure && !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    if UsingAArch32() then
        (validEL, mpamEL) = ELFromM32(PSTATE.M);
    else
        mpamEL = PSTATE.EL;
        validEL = TRUE;
    case acctype of
        when AccType_IFETCH, AccType_IC
            InD = TRUE;
        otherwise
            // other access types are DATA accesses
            InD = FALSE;
    if !validEL then
        return pspace = PARTIDspaceFromSS(security);
    if !validEL then
        return DefaultMPAMinfo(pspace);
    if HaveEMPAMExt() && pspace == PIdSpace_Secure then
        if MPAM3_EL3.FORCE_NS == '1' then
            pspace = PIdSpace_NonSecure;
        if MPAM3_EL3.SDEFLT == '1' then
            return DefaultMPAMinfo(pspace);
    if !MPAMisEnabled() then
        return;
    if MPAM3_EL3.SDEFLT == '1' then
        return DefaultMPAMinfo(pspace);
    else
        return return genMPAM(mpamEL, InD, pspace);
```

Library pseudocode for shared/functions/mpam/MAP_vPARTID

```
// MAP_vPARTID()
// =====
// MAP_vPARTID
// =====
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
// should not ever be called if EL2 is not implemented
// or is implemented but not enabled in the current
// security state.
PARTIDtype ret;
boolean err;
integer virt    = UInt(vpartid);
integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

// vpartid_max is largest vpartid supported
integer vpartid_max = (vpmrmax << 2) + 3;

// One of many ways to reduce vpartid to value less than vpartid_max.
if UInt(vpartid) > vpartid_max then
    virt = virt MOD (vpartid_max+1);

// Check for valid mapping entry.
if MPAMVPMV_EL2<virt> == '1' then
    // vpartid has a valid mapping so access the map.
    ret = mapvpmw(virt);
    err = FALSE;

// Is the default virtual PARTID valid?
elseif MPAMVPMV_EL2<0> == '1' then
    // Yes, so use default mapping for vpartid == 0.
    ret = MPAMVPM0_EL2<0 +: 16>;
    err = FALSE;

// Neither is valid so use default physical PARTID.
else
    ret = DefaultPARTID;
    err = TRUE;

// Check that the physical PARTID is in-range.
// This physical PARTID came from a virtual mapping entry.
integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
if UInt(ret) > partid_max then
    // Out of range, so return default physical PARTID
    ret = DefaultPARTID;
    err = TRUE;
return (ret, err);
```

Library pseudocode for shared/functions/mpam/MPAMisEnabled

```
// MPAMisEnabled()
// =====
// MPAMisEnabled
// =====
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
el = HighestEL();
case el of
    when EL3 return MPAM3_EL3.MPAMEN == '1';
    when EL2 return MPAM2_EL2.MPAMEN == '1';
    when EL1 return MPAM1_EL1.MPAMEN == '1';
```

Library pseudocode for shared/functions/mpam/MPAMisVirtual

```
// MPAMisVirtual()
// =====
// MPAMisVirtual
// =====
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(bits(2) el)
    return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled\(\) &&
            ((el == EL0 && MPAMHCR_EL2.EL0_VPMEN == '1' &&
              (HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0')) ||
             (el == EL1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
```

Library pseudocode for shared/functions/mpam/PARTIDspaceFromSS

```
// PARTIDspaceFromSS()
// =====
// PARTIDspaceFromSS
// =====
// Returns the primary PARTID space from the Security State.

PARTIDspaceType PARTIDspaceFromSS(SecurityState security)
    case security of
        when SS\_NonSecure
            return PIdSpace\_NonSecure;
        when SS\_Secure
            if HaveEMPAMExt\(\) && MPAM3_EL3.FORCE_NS == '1' then
                return PIdSpace\_NonSecure;
            else
                return PIdSpace\_Secure;
        otherwise
            Unreachable\(\);
```

Library pseudocode for shared/functions/mpam/genMPAM

```
// genMPAM()
// =====
// genMPAM
// =====
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// Produces a PARTID in PARTID space pspace.

MPAMinfo genMPAM(bits(2) el, boolean InD, PARTIDspaceType pspace)
    MPAMinfo returninfo;
    PARTIDtype partidel;
    boolean perr;
    // gstplk is guest OS application locked by the EL2 hypervisor to
    // only use EL1 the virtual machine's PARTIDs.
    boolean gstplk = (el == EL0 && EL2Enabled\(\) &&
                     MPAMHCR_EL2.GSTAPP_PLK == '1' &&
                     HCR_EL2.TGE == '0');
    bits(2) eff_el = if gstplk then EL1 else el;
    (partidel, perr) = genPARTID(eff_el, InD);
    PMGtype groupel = genPMG(eff_el, InD, perr);
    returninfo.mpam_ns = pspace;
    returninfo.partid = partidel;
    returninfo.pmg = groupel;
    return returninfo;
```

Library pseudocode for shared/functions/mpam/genMPAMel

```
// genMPAMel()
// =====
// genMPAMel
// =====
// Returns MPAMinfo for specified EL in the current security state.
// InD is TRUE for instruction access and FALSE otherwise.

MPAMinfo genMPAMel(bits(2) el, boolean InD)
    SecurityState security = SecurityStateAtEL(el);
    PARTIDspaceType space = PARTIDspaceFromSS(security);
    boolean use_default = !(HaveMPAMExt() && MPAMisEnabled());
    if HaveEMPAMExt() && space == PIdSpace_Secure then
        if MPAM3_EL3.FORCE_NS == '1' then
            space = PIdSpace_NonSecure;
        if MPAM3_EL3.SDEFLT == '1' then
            use_default = TRUE;
    if !use_default then
        return genMPAM(el, InD, space);
    else
        return DefaultMPAMinfo(space);
```

Library pseudocode for shared/functions/mpam/genPARTID

```
// genPARTID()
// =====
// genPARTID
// =====
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.

(PARTIDtype, boolean) genPARTID(bits(2) el, boolean InD)
    PARTIDtype partidel = getMPAM_PARTID(el, InD);
    PARTIDtype partid_max = MPAMIDR_EL1.PARTID_MAX;
    if UInt(partidel) > UInt(partid_max) then
        return (DefaultPARTID, TRUE);
    if MPAMisVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```

Library pseudocode for shared/functions/mpam/genPMG

```
// genPMG()
// =====
// genPMG
// =====
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(bits(2) el, boolean InD, boolean partid_err)
    integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);
    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DefaultPMG;
    PMGtype groupel = getMPAM_PMG(el, InD);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DefaultPMG;
```

Library pseudocode for shared/functions/mpam/getMPAM_PARTID

```
// getMPAM_PARTID()
// =====
// getMPAM_PARTID
// =====
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(bits(2) MPAMn, boolean InD)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_I;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros();
            when '01' partid = MPAM1_EL1.PARTID_I;
            when '00' partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
        else
            case MPAMn of
                when '11' partid = MPAM3_EL3.PARTID_D;
                when '10' partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros();
                when '01' partid = MPAM1_EL1.PARTID_D;
                when '00' partid = MPAM0_EL1.PARTID_D;
                otherwise partid = PARTIDtype UNKNOWN;
            return partid;
```

Library pseudocode for shared/functions/mpam/getMPAM_PMG

```
// getMPAM_PMG()
// =====
// getMPAM_PMG
// =====
// Returns a PMG from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(bits(2) MPAMn, boolean InD)
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_I;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros();
            when '01' pmg = MPAM1_EL1.PMG_I;
            when '00' pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
        else
            case MPAMn of
                when '11' pmg = MPAM3_EL3.PMG_D;
                when '10' pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros();
                when '01' pmg = MPAM1_EL1.PMG_D;
                when '00' pmg = MPAM0_EL1.PMG_D;
                otherwise pmg = PMGtype UNKNOWN;
            return pmg;
```

Library pseudocode for shared/functions/mpam/mapvpmw

```
// mapvpmw()
// =====
// mapvpmw
// =====
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros(64);
    // vpme_lsb selects LSB of field within register
    integer vpme_lsb = (vpartid MOD 4) * 16;
    return vpmw<vpme_lsb+: 16>;
```

Library pseudocode for shared/functions/registers/BranchTo

```
// BranchTo()
// =====
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type, boolean branch_conditional)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>);
        _PC = target_vaddress;
    return;
```

Library pseudocode for shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// =====
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

Library pseudocode for shared/functions/registers/BranchType

```
enumeration BranchType {
    BranchType_DIRCALL,    // Direct Branch with link
    BranchType_INDCALL,    // Indirect Branch with link
    BranchType_ERET,       // Exception return (indirect)
    BranchType_DBGEXIT,    // Exit from Debug state
    BranchType_RET,        // Indirect branch with function return hint
    BranchType_DIR,        // Direct branch
    BranchType_INDIR,      // Indirect branch
    BranchType_EXCEPTION,  // Exception entry
    BranchType_RESET,      // Reset
    BranchType_UNKNOWN};   // Other
```

Library pseudocode for shared/functions/registers/Hint_Branch

```
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.
Hint_Branch(BranchType hint);
```

Library pseudocode for shared/functions/registers/NextInstrAddr

```
// Return address of the sequentially next instruction.
bits(N) NextInstrAddr();
```

Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

```
// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);
```

Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// =====
// Return address of the current instruction.

bits(N) ThisInstrAddr()
    assert N == 64 || (N == 32 && UsingAArch32\(\));
    return _PC<N-1:0>;
```

Library pseudocode for shared/functions/registers/_PC

```
bits(64) _PC;
```

Library pseudocode for shared/functions/registers/_R

```
array bits(64) _R[0..30];
```


Library pseudocode for shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// =====

bits(N) SPSR[]
  bits(N) result;
  if UsingAArch32() then
    assert N == 32;
    case PSTATE.M of
      when M32_FIQ      result = SPSR_fiq<N-1:0>;
      when M32_IRQ      result = SPSR_irq<N-1:0>;
      when M32_Svc      result = SPSR_svc<N-1:0>;
      when M32_Monitor  result = SPSR_mon<N-1:0>;
      when M32_Abort    result = SPSR_abt<N-1:0>;
      when M32_Hyp      result = SPSR_hyp<N-1:0>;
      when M32_Undef    result = SPSR_und<N-1:0>;
      otherwise         Unreachable();
  else
    assert N == 64;
    case PSTATE.EL of
      when EL1          result = SPSR_EL1<N-1:0>;
      when EL2          result = SPSR_EL2<N-1:0>;
      when EL3          result = SPSR_EL3<N-1:0>;
      otherwise         Unreachable();
  return result;

// SPSR[] - assignment form
// =====

SPSR[] = bits(N) value
  if UsingAArch32() then
    assert N == 32;
    case PSTATE.M of
      when M32_FIQ      SPSR_fiq = ZeroExtend(value);
      when M32_IRQ      SPSR_irq = ZeroExtend(value);
      when M32_Svc      SPSR_svc = ZeroExtend(value);
      when M32_Monitor  SPSR_mon = ZeroExtend(value);
      when M32_Abort    SPSR_abt = ZeroExtend(value);
      when M32_Hyp      SPSR_hyp = ZeroExtend(value);
      when M32_Undef    SPSR_und = ZeroExtend(value);
      otherwise         Unreachable();
  else
    assert N == 64;
    case PSTATE.EL of
      when EL1          SPSR_EL1 = ZeroExtend(value);
      when EL2          SPSR_EL2 = ZeroExtend(value);
      when EL3          SPSR_EL3 = ZeroExtend(value);
      otherwise         Unreachable();
  return;
```

Library pseudocode for shared/functions/system/ArchVersion

```
enumeration ArchVersion {
  ARMv8p0
  , ARMv8p1
  , ARMv8p2
  , ARMv8p3
  , ARMv8p4
  , ARMv8p5
  , ARMv8p6
  , ARMv8p7
};
```

Library pseudocode for shared/functions/system/BranchTargetCheck

```
// BranchTargetCheck()
// =====
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

BranchTargetCheck()
    assert HaveBTIExt() && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
        bits(64) pc = ThisInstrAddr();
        AArch64.BranchTargetException(pc<51:0>);

    boolean branch_instr = AArch64.ExecutingBR0rBLR0rRetInstr();
    boolean bti_instr    = AArch64.ExecutingBTIIInstr();

    // PSTATE.BTYPE defaults to 00 for instructions that do not explicitly set BTYPE.
    if !(branch_instr || bti_instr) then
        BTypeNext = '00';
```

Library pseudocode for shared/functions/system/ClearEventRegister

```
// ClearEventRegister()
// =====
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;
```

Library pseudocode for shared/functions/system/ClearPendingPhysicalSError

```
// Clear a pending physical SError interrupt.
ClearPendingPhysicalSError();
```

Library pseudocode for shared/functions/system/ClearPendingVirtualSError

```
// Clear a pending virtual SError interrupt.
ClearPendingVirtualSError();
```

Library pseudocode for shared/functions/system/ConditionHolds

```
// ConditionHolds()
// =====
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
// Evaluate base condition.
case cond<3:1> of
  when '000' result = (PSTATE.Z == '1');           // EQ or NE
  when '001' result = (PSTATE.C == '1');           // CS or CC
  when '010' result = (PSTATE.N == '1');           // MI or PL
  when '011' result = (PSTATE.V == '1');           // VS or VC
  when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
  when '101' result = (PSTATE.N == PSTATE.V);       // GE or LT
  when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
  when '111' result = TRUE;                         // AL

// Condition flag values in the set '111x' indicate always true
// Otherwise, invert condition if necessary.
if cond<0> == '1' && cond != '1111' then
  result = !result;

return result;
```

Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrier

```
ConsumptionOfSpeculativeDataBarrier();
```

Library pseudocode for shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// =====

InstrSet CurrentInstrSet()

if UsingAArch32\(\) then
  result = if PSTATE.T == '0' then InstrSet\_A32 else InstrSet\_T32;
  // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
else
  result = InstrSet\_A64;
return result;
```

Library pseudocode for shared/functions/system/CurrentPL

```
// CurrentPL()
// =====

PrivilegeLevel CurrentPL()
return PLOfEL(PSTATE.EL);
```

Library pseudocode for shared/functions/system/DSBAlias

```
enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
```

Library pseudocode for shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

Library pseudocode for shared/functions/system/EL2Enabled

```
// EL2Enabled()
// =====
// Returns TRUE if EL2 is present and executing
// - with SCR_EL3.NS==1 when Non-secure EL2 is implemented, or
// - with SCR_EL3.NS==0 when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
    return HaveEL(EL2) && (!HaveEL(EL3) || SCR_EL3.NS == '1' || IsSecureEL2Enabled());
```

Library pseudocode for shared/functions/system/ELFromM32

```
// ELFromM32()
// =====

(boolean, bits(2)) ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid, EL):
    //   'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    //   and the current value of SCR.NS/SCR_EL3.NS.
    //   'EL' is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = !BadMode(mode); // Check for modes that are not valid for this implementation
    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
            valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && !HaveAArch64HighestELUsingAArch32() && SCR.NS == '0' then EL3 else EL1);
        when M32_User
            el = EL0;
        otherwise
            valid = FALSE; // Passed an illegal mode value
    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

Library pseudocode for shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// =====

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
// 'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
// 'EL' is the Exception level decoded from 'spsr'.

(boolean, bits(2)) ELFromSPSR(bits(N) spsr)
    if spsr<4> == '0' then // AArch64 state
        el = spsr<3:2>;
        if !HaveAArch64HighestELUsingAArch32() then // No AArch64 support
            valid = FALSE;
        elseif !HaveEL(el) then // Exception level not implemented
            valid = FALSE;
        elseif spsr<1> == '1' then // M[1] must be 0
            valid = FALSE;
        elseif el == EL0 && spsr<0> == '1' then // for EL0, M[0] must be 0
            valid = FALSE;
        elseif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
            valid = FALSE; // Unless Secure EL2 is enabled, EL2 only valid in Non-secure state
        else
            valid = TRUE;
    elseif HaveAArch32HaveAnyAArch32() then // AArch32 state
        (valid, el) = ELFromM32(spsr<4:0>);
    else
        valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

Library pseudocode for shared/functions/system/ELIsInHost

```
// ELIsInHost()
// =====

boolean ELIsInHost(bits(2) el)
    if !HaveVirtHostExt() || ELUsingAArch32(EL2) then
        return FALSE;
    case el of
        when EL3
            return FALSE;
        when EL2
            return HCR_EL2.E2H == '1';
        when EL2Enabled() && HCR_EL2.E2H == '1';
        when EL1
            return FALSE;
        when EL0
            return EL2Enabled() && HCR_EL2.<E2H, TGE> == '11';
        otherwise
            Unreachable();
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1, EL2, EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// =====

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
// Returns (known, aarch32):
// 'known' is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
// using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
// 'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
if !HaveAArch32EL(el) then
    return (TRUE, FALSE); // Exception level is using AArch64
elseif secure && el == EL2 then
    return (TRUE, FALSE); // Secure EL2 is using AArch64
elseif !elseif HaveAArch64HighestELUsingAArch32() then
    return (TRUE, TRUE); // Highest Exception level, and therefore all levels are using AArch64
elseif el == HighestEL() then
    return (TRUE, FALSE); // This is highest Exception level, so is using AArch32

// Remainder of function deals with the interprocessing cases when highest Exception level is using AArch32

boolean aarch32 = boolean UNKNOWN;
boolean known = TRUE;

aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext() || SCR_EL3.EEL2 == '1')
aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) && ((HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1') || !secure) && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt()))
// Only know if EL0 using AArch32 from PSTATE
if el == EL0 && !aarch32_at_el1 then
    if PSTATE.EL == EL0 then
        aarch32 = PSTATE.nRW == '1'; // EL0 controlled by PSTATE
    else
        known = FALSE; // EL0 state is UNKNOWN
else
    aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1, EL0});

if !known then aarch32 = boolean UNKNOWN;
return (known, aarch32);
```

Library pseudocode for shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// =====

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =====

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/EndOfInstruction

```
// Terminate processing of the current instruction.
EndOfInstruction();
```

Library pseudocode for shared/functions/system/EnterLowPowerState

```
// PE enters a low-power state.
EnterLowPowerState();
```

Library pseudocode for shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

Library pseudocode for shared/functions/system/ExceptionalOccurrenceTargetState

```
enumeration ExceptionalOccurrenceTargetState {  
    AArch32_NonDebugState,  
    AArch64_NonDebugState,  
    DebugState  
};
```

Library pseudocode for shared/functions/system/FIQPending

```
// Returns TRUE if there is any pending physical FIQ.  
boolean FIQPending();
```

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()  
// =====  
// Return a PSR value which represents the current PSTATE  
  
bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState)  
    if UsingAArch32() && (targetELState IN {AArch32_NonDebugState, DebugState}) then  
        assert N == 32;  
    else  
        assert N == 64;  
        bits(N) spsr = Zeros();  
        spsr<31:28> = PSTATE.<N,Z,C,V>;  
        if HavePANExt() then spsr<22> = PSTATE.PAN;  
        spsr<20> = PSTATE.IL;  
        if PSTATE.nRW == '1' then // AArch32 state  
            spsr<27> = PSTATE.Q;  
            spsr<26:25> = PSTATE.IT<1:0>;  
            if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;  
            if HaveDITExt() then  
                if targetELState == AArch32_NonDebugState then  
                    spsr<21> = PSTATE.DIT;  
                else // AArch64_NonDebugState or DebugState  
                    spsr<24> = PSTATE.DIT;  
            if targetELState IN {AArch64_NonDebugState, DebugState} then  
                spsr<21> = PSTATE.SS;  
            spsr<19:16> = PSTATE.GE;  
            spsr<15:10> = PSTATE.IT<7:2>;  
            spsr<9> = PSTATE.E;  
            spsr<8:6> = PSTATE.<A,I,F>; // No PSTATE.D in AArch32 state  
            spsr<5> = PSTATE.T;  
            assert PSTATE.M<4> == PSTATE.nRW; // bit [4] is the discriminator  
            spsr<4:0> = PSTATE.M;  
        else // AArch64 state  
            if HaveMTEExt() then spsr<25> = PSTATE.TC0;  
            if HaveDITExt() then spsr<24> = PSTATE.DIT;  
            if HaveUA0Ext() then spsr<23> = PSTATE.UA0;  
            spsr<21> = PSTATE.SS;  
            if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;  
            if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;  
            spsr<9:6> = PSTATE.<D,A,I,F>;  
            spsr<4> = PSTATE.nRW;  
            spsr<3:2> = PSTATE.EL;  
            spsr<0> = PSTATE.SP;  
        return spsr;
```

Library pseudocode for shared/functions/system/HasArchVersion

```
// HasArchVersion()
// =====
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAArch32HaveAArch32EL

```
// HaveAArch32()
// =====
// Return TRUE if AArch32 state is supported at at least EL0.
// HaveAArch32EL()
// =====

boolean HaveAArch32()
    return boolean IMPLEMENTATION_DEFINED; HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE; // The Exception level is not implemented
    elseif !HaveAnyAArch32() then
        return FALSE; // No Exception level can use AArch32
    elseif HighestELUsingAArch32() then
        return TRUE; // All Exception levels are using AArch32
    elseif el == HighestEL() then
        return FALSE; // The highest Exception level is using AArch64
    elseif el == EL0 then
        return TRUE; // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAArch32ELHaveAnyAArch32

```
// HaveAArch32EL()
// =====
// HaveAnyAArch32()
// =====
// Return TRUE if AArch32 state is supported at any Exception level.

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveAnyAArch32()
        return boolean IMPLEMENTATION_DEFINED; HaveEL(el) then
        return FALSE; // The Exception level is not implemented
    elseif !HaveAArch32() then
        return FALSE; // No Exception level can use AArch32
    elseif !HaveAArch64() then
        return TRUE; // All Exception levels are using AArch32
    elseif el == HighestEL() then
        return FALSE; // The highest Exception level is using AArch64
    elseif el == EL0 then
        return TRUE; // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAArch64HaveAnyAArch64

```
// HaveAArch64()
// =====
// Return TRUE if AArch64 state is supported at the highest Exception level.
// HaveAnyAArch64()
// =====
// Return TRUE if AArch64 state is supported at any Exception level.

boolean HaveAArch64()
    return boolean IMPLEMENTATION_DEFINED "Highest EL using AArch64"; HaveAnyAArch64()
    return !HighestELUsingAArch32();
```


Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// =====
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    if el IN {EL1, EL0} then
        return TRUE; // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// =====
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when EL3
            assert secure;
            return HaveEL(EL3);
        when EL2
            if secure then
                return HaveEL(EL2) && HaveSecureEL2Ext();
            else
                return HaveEL(EL2);
        otherwise
            return (HaveEL(EL3) ||
                (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

Library pseudocode for shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =====
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HighestEL

```
// HighestEL()
// =====
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

Library pseudocode for shared/functions/system/HighestELUsingAArch32

```
// HighestELUsingAArch32()
// =====
// Return TRUE if the highest implemented Exception level is using AArch32

boolean HighestELUsingAArch32()
    if !HaveAnyAArch32() then return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Highest EL using AArch32";
```

Library pseudocode for shared/functions/system/Hint_DGH

```
// Provides a hint to close any gathering occurring within the micro-architecture.  
Hint_DGH();
```

Library pseudocode for shared/functions/system/Hint_WFE

```
// Hint_WFE()  
// =====  
// Provides a hint indicating that the PE can enter a low-power state  
// and remain there until a wakeup event occurs or, for WFET, a local  
// timeout event is generated when the virtual timer value equals or  
// exceeds the supplied threshold value.  
  
Hint_WFE(integer localtimeout, WfxType wfxtype)  
    if IsEventRegisterSet() then  
        ClearEventRegister();  
    else  
        trap = FALSE;  
        if PSTATE.EL == EL0 then  
            // Check for traps described by the OS which may be EL1 or EL2.  
            if HaveTWEExt() then  
                sctlr = SCTLR[];  
                trap = sctlr.nTWE == '0';  
                target_el = EL1;  
            else  
                AArch64.CheckForWfxTrap(EL1, wfxtype);  
        if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then  
            // Check for traps described by the Hypervisor.  
            if HaveTWEExt() then  
                trap = HCR_EL2.TWE == '1';  
                target_el = EL2;  
            else  
                AArch64.CheckForWfxTrap(EL2, wfxtype);  
  
        if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then  
            // Check for traps described by the Secure Monitor.  
            if HaveTWEExt() then  
                trap = SCR_EL3.TWE == '1';  
                target_el = EL3;  
            else  
                AArch64.CheckForWfxTrap(EL3, wfxtype);  
  
        if trap && PSTATE.EL != EL3 then  
            (delay_enabled, delay) = WFETrapDelay(target_el); // (If trap delay is enabled, Delay amount)  
            if !WaitForEventUntilDelay(delay_enabled, delay) then  
                // Event did not arrive before delay expired  
                AArch64.WfxTrap(wfxtype, target_el); // Trap WFE  
    else  
        WaitForEvent(localtimeout);
```

Library pseudocode for shared/functions/system/Hint_WFI

```
// Hint_WFI()
// =====
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

Hint_WFI(integer localtimeout, WfxType wfxtype)
    if !InterruptPending() then
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch64.CheckForWfxTrap(EL1, wfxtype);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWfxTrap(EL2, wfxtype);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWfxTrap(EL3, wfxtype);
        WaitForInterrupt(localtimeout);
```

Library pseudocode for shared/functions/system/Hint_Yield

```
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.
Hint_Yield();
```

Library pseudocode for shared/functions/system/IRQPending

```
// Returns TRUE if there is any pending physical IRQ.
boolean IRQPending();
```

Library pseudocode for shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// =====

boolean IllegalExceptionReturn(bits(N) spsr)

    // Check for illegal return:
    // * To an unimplemented Exception level.
    // * To EL2 in Secure state, when SecureEL2 is not enabled.
    // * To EL0 using AArch64 state, with SPSR.M[0]==1.
    // * To AArch64 state with SPSR.M[1]==1.
    // * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    // * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //   Execution state used in the Exception level being returned to, as determined by
    //   the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    // * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //   SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    // * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
    return FALSE;
```

Library pseudocode for shared/functions/system/InstrSet

```
enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

```
InstructionSynchronizationBarrier();
```

Library pseudocode for shared/functions/system/InterruptPending

```
// InterruptPending()
// =====
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
    boolean pending_virtual_interrupt = FALSE;
    boolean pending_physical_interrupt = (bit vIRQstatus = (if VirtualIRQPending() then '1' else '0')
    bit vFIQstatus = (if VirtualFIQPending() then '1' else '0') OR HCR_EL2.VF;
    bits(3) v_interrupts = HCR_EL2.VSE : vIRQstatus : vFIQstatus;

    pending_physical_interrupt = (IRQPending() || FIQPending() ||
                                IsPhysicalSErrorPending());

    if pending_virtual_interrupt = ! EL2EnabledIsInHost() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE =
        boolean virq_pending = HCR_EL2.IMO == '1' && (VirtualIRQPending() || HCR_EL2.VI == '1');
        boolean vfiq_pending = HCR_EL2.FMO == '1' && (VirtualFIQPending() || HCR_EL2.VF == '1');
        boolean vsei_pending = HCR_EL2.AMO == '1' && (IsVirtualSErrorPending() || HCR_EL2.VSE == '1');
        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    ) && ((v_interrupts AND
        HCR_EL2.<AMO,IMO,FMO>) != '000');
    return pending_physical_interrupt || pending_virtual_interrupt;
```

Library pseudocode for shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// =====
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

Library pseudocode for shared/functions/system/IsHighestEL

```
// IsHighestEL()
// =====
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;
```

Library pseudocode for shared/functions/system/IsInHost

```
// IsInHost()
// =====

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

Library pseudocode for shared/functions/system/IsPhysicalSErrorPending

```
// Returns TRUE if a physical SError interrupt is pending.
boolean IsPhysicalSErrorPending();
```

Library pseudocode for shared/functions/system/IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()
// =====
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered(bits(2) target_el, bits(25) syndrome)
    if HaveRASExt() then
        if HaveDoubleFaultExt() then
            return TRUE;

        if ELUsingAArch32(target_el) then
            if syndrome<11:10> != '00' then
                // AArch32 and not Uncontainable.
                return TRUE;
        else
            if syndrome<24> == '0' && syndrome<5:0> != '000000' then
                // AArch64 and neither IMPLEMENTATION_DEFINED syndrome nor Uncategorized.
                return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

Library pseudocode for shared/functions/system/IsSecure

```
// IsSecure()
// =====
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elseif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
```

Library pseudocode for shared/functions/system/IsSecureBelowEL3

```
// IsSecureBelowEL3()
// =====
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL(EL3) then
        return SCR_GEN[].NS == '0';
    elseif HaveEL(EL2) && (!HaveSecureEL2Ext() || !() || HaveAArch64HighestELUsingAArch32()) then
        // If Secure EL2 is not an architecture option then we must be Non-secure.
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

Library pseudocode for shared/functions/system/IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// =====
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    if HaveEL(EL2) && HaveSecureEL2Ext() then
        if HaveEL(EL3) then
            if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
        else
            return IsSecure();
    else
        return FALSE;
```

Library pseudocode for shared/functions/system/IsSynchronizablePhysicalSErrorPending

```
// Returns TRUE if a synchronizable physical SError interrupt is pending.
boolean IsSynchronizablePhysicalSErrorPending();
```

Library pseudocode for shared/functions/system/IsVirtualSErrorPending

```
// Returns TRUE if a virtual SError interrupt is pending.
boolean IsVirtualSErrorPending();
```

Library pseudocode for shared/functions/system/LocalTimeoutEvent

```
// Returns TRUE if a local timeout event is generated when the value of
// CNTVCT_EL0 equals or exceeds the threshold value for the first time.
// If the threshold value is less than zero a local timeout event will
// not be generated.
boolean LocalTimeoutEvent(integer localtimeout);
```

Library pseudocode for shared/functions/system/Mode_Bits

```
constant bits(5) M32_User      = '10000';
constant bits(5) M32_FIQ      = '10001';
constant bits(5) M32_IRQ      = '10010';
constant bits(5) M32_Svc      = '10011';
constant bits(5) M32_Monitor  = '10110';
constant bits(5) M32_Abort     = '10111';
constant bits(5) M32_Hyp       = '11010';
constant bits(5) M32_Undef     = '11011';
constant bits(5) M32_System    = '11111';
```

Library pseudocode for shared/functions/system/PL0fEL

```
// PL0fEL()
// =====

PrivilegeLevel PL0fEL(bits(2) el)
    case el of
        when EL3 return if !return if HaveAArch64HighestELUsingAArch32() then PL1 else PL3;
        when EL2 return PL2;
        when EL1 return PL1;
        when EL0 return PL0;
```

Library pseudocode for shared/functions/system/PSTATE

```
ProcState PSTATE;
```

Library pseudocode for shared/functions/system/PhysicalCountInt

```
// PhysicalCountInt()
// =====
// Returns the integral part of physical count value of the System counter.

bits(64) PhysicalCountInt()
return PhysicalCount<87:24>;PhysicalCountInt();
```

Library pseudocode for shared/functions/system/PrivilegeLevel

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

Library pseudocode for shared/functions/system/ProcState

```
type ProcState is (
    bits (1) N,           // Negative condition flag
    bits (1) Z,           // Zero condition flag
    bits (1) C,           // Carry condition flag
    bits (1) V,           // oVerflow condition flag
    bits (1) D,           // Debug mask bit [AArch64 only]
    bits (1) A,           // SError interrupt mask bit
    bits (1) I,           // IRQ mask bit
    bits (1) F,           // FIQ mask bit
    bits (1) PAN,         // Privileged Access Never Bit [v8.1]
    bits (1) UAO,         // User Access Override [v8.2]
    bits (1) DIT,         // Data Independent Timing [v8.4]
    bits (1) TCO,         // Tag Check Override [v8.5, AArch64 only]
    bits (2) BTYPE,       // Branch Type [v8.5]
    bits (1) SS,          // Software step bit
    bits (1) IL,          // Illegal Execution state bit
    bits (2) EL,          // Exception level
    bits (1) nRW,         // not Register Width: 0=64, 1=32
    bits (1) SP,          // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,           // Cumulative saturation flag [AArch32 only]
    bits (4) GE,          // Greater than or Equal flags [AArch32 only]
    bits (1) SSBS,        // Speculative Store Bypass Safe
    bits (8) IT,          // If-then bits, RES0 in CPSR [AArch32 only]
    bits (1) J,           // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,           // T32 bit, RES0 in CPSR [AArch32 only]
    bits (1) E,           // Endianness bit [AArch32 only]
    bits (5) M            // Mode field [AArch32 only]
)
```


Library pseudocode for shared/functions/system/RestoredITBits

```
// RestoredITBits()
// =====
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool\(Unpredictable\_ILZEROIT\) then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero\(it<7:4>\) && IsZero\(it<3:0>\) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLRL.ITD;
    if (spsr<5> == '0' && !IsZero\(it\)) || (itd == '1' && !IsZero\(it<2:0>\)) then
        return '00000000';
    else
        return it;
```

Library pseudocode for shared/functions/system/SCRType

```
type SCRType;
```

Library pseudocode for shared/functions/system/SCR_GEN

```
// SCR_GEN[]
// =====

SCRType SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL\(EL3\);
    bits(64) r;
    if !ifHaveAArch64HighestELUsingAArch32\(\) then
        r = ZeroExtend\(SCR\);
    else
        r = SCR_EL3;
    return r;
```

Library pseudocode for shared/functions/system/SecurityState

```
enumeration SecurityState {
    SS_NonSecure,
    SS_Secure
};
```

Library pseudocode for shared/functions/system/SendEvent

```
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.
SendEvent();
```

Library pseudocode for shared/functions/system/SendEventLocal

```
// SendEventLocal()
// =====
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed.

SendEventLocal()
    EventRegister = '1';
    return;
```

Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// =====
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(N) spsr)
    boolean from_aarch64 = !UsingAArch32();
    assert N == (if from_aarch64 then 64 else 32);
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    ShouldAdvanceSS = FALSE;
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveBTIExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
        if HaveDITExt() then PSTATE.DIT = bit UNKNOWN;
        if HaveMTEExt() then PSTATE.TCO = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then // AArch32 state
            AArch32.WriteMode(spsr<4:0>; // Sets PSTATE.EL correctly
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
        else // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL = spsr<3:2>;
            PSTATE.SP = spsr<0>;
            if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
            if HaveUAOExt() then PSTATE.UAO = spsr<23>;
            if HaveDITExt() then PSTATE.DIT = spsr<24>;
            if HaveMTEExt() then PSTATE.TCO = spsr<25>;

        // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
        // the T bit is set to zero or copied from SPsr.
        if PSTATE.IL == '1' && PSTATE.nRW == '1' then
            if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';

        // State that is reinstated regardless of illegal exception return
        PSTATE.<N,Z,C,V> = spsr<31:28>;
        if HavePANExt() then PSTATE.PAN = spsr<22>;
        if PSTATE.nRW == '1' then // AArch32 state
            PSTATE.Q = spsr<27>;
            PSTATE.IT = RestoredITBits(spsr);
            ShouldAdvanceIT = FALSE;
            if HaveDITExt() then PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr<24> else spsr<21>);
            PSTATE.GE = spsr<19:16>;
            PSTATE.E = spsr<9>;
            PSTATE.<A,I,F> = spsr<8:6>; // No PSTATE.D in AArch32 state
            PSTATE.T = spsr<5>; // PSTATE.J is RES0
        else // AArch64 state
            // No PSTATE.<Q,IT,GE,E,T> in AArch64 state
            PSTATE.<D,A,I,F> = spsr<9:6>;
    return;
```

Library pseudocode for shared/functions/system/ShouldAdvanceIT

```
boolean ShouldAdvanceIT;
```

Library pseudocode for shared/functions/system/ShouldAdvanceSS

```
boolean ShouldAdvanceSS;
```

Library pseudocode for shared/functions/system/SpeculationBarrier

```
SpeculationBarrier();
```

Library pseudocode for shared/functions/system/SynchronizeContext

```
SynchronizeContext();
```

Library pseudocode for shared/functions/system/SynchronizeErrors

```
// Implements the error synchronization event.  
SynchronizeErrors();
```

Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt.  
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt or unmasked virtual SError  
// interrupt.  
TakeUnmaskedSErrorInterrupts();
```

Library pseudocode for shared/functions/system/ThisInstr

```
bits(32) ThisInstr();
```

Library pseudocode for shared/functions/system/ThisInstrLength

```
integer ThisInstrLength();
```

Library pseudocode for shared/functions/system/Unreachable

```
Unreachable()  
    assert FALSE;
```

Library pseudocode for shared/functions/system/UsingAArch32

```
// UsingAArch32()  
// =====  
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.  
  
boolean UsingAArch32()  
    boolean aarch32 = (PSTATE.nRW == '1');  
    if !HaveAArch32HaveAnyAArch32() then assert !aarch32;  
    if !if HaveAArch64HighestELUsingAArch32() then assert aarch32;  
    return aarch32;
```

Library pseudocode for shared/functions/system/VirtualFIQPending

```
// Returns TRUE if there is any pending virtual FIQ.  
boolean VirtualFIQPending();
```

Library pseudocode for shared/functions/system/VirtualIRQPending

```
// Returns TRUE if there is any pending virtual IRQ.  
boolean VirtualIRQPending();
```

Library pseudocode for shared/functions/system/WFxType

```
enumeration WFxType {WfxType_WFE, WfxType_WFI, WfxType_WFET, WfxType_WFIT};
```

Library pseudocode for shared/functions/system/WaitForEvent

```
// WaitForEvent()  
// =====  
// PE optionally suspends execution until one of the following occurs:  
// - A WFE wake-up event.  
// - A reset.  
// - The implementation chooses to resume execution.  
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs  
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of  
// suspension causes the Event Register to be cleared.  
  
WaitForEvent(integer localtimeout)  
    if !(IsEventRegisterSet() || LocalTimeoutEvent(localtimeout)) then  
        EnterLowPowerState();  
    return;
```

Library pseudocode for shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()  
// =====  
// PE optionally suspends execution until one of the following occurs:  
// - A WFI wake-up event.  
// - A reset.  
// - The implementation chooses to resume execution.  
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.  
  
WaitForInterrupt(integer localtimeout)  
    if localtimeout < 0 then  
        EnterLowPowerState();  
    else  
        if !LocalTimeoutEvent(localtimeout) then  
            EnterLowPowerState();  
    return;
```



```
// ConstrainUnpredictable()
// =====
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.
```

```
Constraint ConstrainUnpredictable(Unpredictable which)
case which of
    when Unpredictable_VMSR
        return Constraint_UNDEF;
    when Unpredictable_WBOVERLAPLD
        return Constraint_WBSUPPRESS; // return loaded value
    when Unpredictable_WBOVERLAPST
        return Constraint_NONE; // store pre-writeback value
    when Unpredictable_LDPOVERLAP
        return Constraint_UNDEF; // instruction is UNDEFINED
    when Unpredictable_BASEOVERLAP
        return Constraint_UNKNOWNConstraint_NONE; // use UNKNOWN address
; // use original address
    when Unpredictable_DATAOVERLAP
        return Constraint_UNKNOWNConstraint_NONE; // store UNKNOWN value
; // store original value
    when Unpredictable_DEVPAGE2
        return Constraint_FAULT; // take an alignment fault
    when Unpredictable_DEVICETAGSTORE
        return Constraint_NONE; // Do not take a fault
    when Unpredictable_INSTRDEVICE
        return Constraint_NONE; // Do not take a fault
    when Unpredictable_RESCPACR
        return Constraint_TRUE; // Map to UNKNOWN value
    when Unpredictable_RESMAIR
        return Constraint_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable_S1CTAGGED
        return Constraint_FALSE; // SCTLx_ELx.C == '0' marks address as untagged
    when Unpredictable_S2RESMEMATTR
        return Constraint_NC; // Map to Noncacheable value
    when Unpredictable_RESTEXCB
        return Constraint_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable_RESDACR
        return Constraint_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable_RESPRRR
        return Constraint_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable_RESVTCRS
        return Constraint_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable_RESTnSZ
        return Constraint_FORCE; // Map to the limit value
    when Unpredictable_OORTnSZ
        return Constraint_FORCE; // Map to the limit value
    when Unpredictable_LARGEIPA
        return Constraint_FORCE; // Restrict the IA size to the PAMax value
    when Unpredictable_ESRCONDPASS
        return Constraint_FALSE; // Report as "AL"
    when Unpredictable_ILZEROIT
        return Constraint_FALSE; // Do not zero PSTATE.IT
    when Unpredictable_ILZEROT
        return Constraint_FALSE; // Do not zero PSTATE.T
    when Unpredictable_BPVECTORCATCHPRI
        return Constraint_TRUE; // Debug Vector Catch: match on 2nd halfword
    when Unpredictable_VCMATCHHALF
        return Constraint_FALSE; // No match
    when Unpredictable_VCMATCHDAPA
        return Constraint_FALSE; // No match on Data Abort or Prefetch abort
    when Unpredictable_WPMASKANDBAS
        return Constraint_FALSE; // Watchpoint disabled
```

```

when Unpredictable_WPBASCONTIGUOUS
    return Constraint_FALSE;    // Watchpoint disabled
when Unpredictable_RESWPMASK
    return Constraint_DISABLED; // Watchpoint disabled
when Unpredictable_WPMASKEDBITS
    return Constraint_FALSE;    // Watchpoint disabled
when Unpredictable_RESBPWPCTRL
    return Constraint_DISABLED; // Breakpoint/watchpoint disabled
when Unpredictable_BPNOTIMPL
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_RESBPTYPE
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPNOTCTXCMP
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPMATCHHALF
    return Constraint_FALSE;    // No match
when Unpredictable_BPMISMATCHHALF
    return Constraint_FALSE;    // No match
when Unpredictable_RESTARTALIGNPC
    return Constraint_FALSE;    // Do not force alignment
when Unpredictable_RESTARTZEROUPPERPC
    return Constraint_TRUE;     // Force zero extension
when Unpredictable_ZEROUPPER
    return Constraint_TRUE;     // zero top halves of X registers
when Unpredictable_ERETZEROUPPERPC
    return Constraint_TRUE;     // zero top half of PC
when Unpredictable_A32FORCEALIGNPC
    return Constraint_FALSE;    // Do not force alignment
when Unpredictable_SMD
    return Constraint_UNDEF;    // disabled SMC is Unallocated
when Unpredictable_NONFAULT
    return Constraint_FALSE;    // Speculation enabled
when Unpredictable_SVEZEROUPPER
    return Constraint_TRUE;     // zero top bits of Z registers
when Unpredictable_SVELDNFDATA
    return Constraint_TRUE;     // Load mem data in NF loads
when Unpredictable_SVELDNFZERO
    return Constraint_TRUE;     // Write zeros in NF loads
when Unpredictable_CHECKSPNONEACTIVE
    return Constraint_TRUE;     // Check SP alignment
when Unpredictable_NVNV1
    return Constraint_NVNV1_00; // Map unpredictable configuration of HCR_EL2<NV,NV1>
                                // to NV = 0 and NV1 = 0
when Unpredictable_Shareability
    return Constraint_OSH;      // Map reserved encoding of shareability to outer shareable
when Unpredictable_AFUPDATE
    return Constraint_TRUE;     // AF update for alignment or permission fault
when Unpredictable_IESBinDebug
    return Constraint_TRUE;     // Use SCTLR[].IESB in Debug state
when Unpredictable_BADPMSFCR
    return Constraint_TRUE;     // Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
when Unpredictable_ZEROBTYP
    return Constraint_TRUE;     // Save BTYPE in SPSR_ELx/DPSR_EL0 as '00'
when Unpredictable_CLEARERRITEZERO
    return Constraint_FALSE;    // Clearing sticky errors when instruction in flight
when Unpredictable_ALUEXCEPTIONRETURN
    return Constraint_UNDEF;
when Unpredictable_DBGxVR_RESS
    return Constraint_FALSE;
when Unpredictable_PMSCR_PCT
    return Constraint_PMSCR_PCT_VIRT;
when Unpredictable_WFXTDEBUG
    return Constraint_FALSE;    // WfXT in Debug state does not execute as a NOP
when Unpredictable_LS64UNSUPPORTED
    return Constraint_LIMITED_ATOMICITY; // Accesses are not single-copy atomic above the byte level
// Misaligned exclusives, atomics, acquire/release to region that is not Normal Cacheable WB are a
when Unpredictable_MISALIGNEDATOMIC
    return Constraint_FALSE;

when Unpredictable_IGNORETRAPINDEBUG

```

```

        return Constraint\_TRUE;    // Trap to register access in debug state is ignored
    when Unpredictable\_PMUEVENTCOUNTER
        return Constraint\_UNDEF;    // Accesses to the register are UNDEFINED

```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

```

// ConstrainUnpredictableBits()
// =====

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint, bits(width)) ConstrainUnpredictableBits(Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint\_UNKNOWN then
        return (c, Zeros(width));    // See notes; this is an example implementation only
    elsif c == Constraint\_PMSCR\_PCT\_VIRT then
        return (c, Zeros(width));
    else
        return (c, bits(width) UNKNOWN);    // bits result not used

```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

```

// ConstrainUnpredictableBool()
// =====

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

boolean ConstrainUnpredictableBool(Unpredictable which)

    c = ConstrainUnpredictable(which);
    assert c IN {Constraint\_TRUE, Constraint\_FALSE};
    return (c == Constraint\_TRUE);

```


Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// =====

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the integer part
// of the result.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high, Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint\_UNKNOWN then
        return (c, low);          // See notes; this is an example implementation only
    else
        return (c, integer UNKNOWN); // integer result not used
```

Library pseudocode for shared/functions/unpredictable/Constraint

```
enumeration Constraint    { // General
    Constraint_NONE,      // Instruction executes with
                          // no change or side-effect to its described b
    Constraint_UNKNOWN,   // Destination register has UNKNOWN value
    Constraint_UNDEF,     // Instruction is UNDEFINED
    Constraint_UNDEFEL0,   // Instruction is UNDEFINED at EL0 only
    Constraint_NOP,       // Instruction executes as NOP
    Constraint_TRUE,
    Constraint_FALSE,
    Constraint_DISABLED,
    Constraint_UNCOND,    // Instruction executes unconditionally
    Constraint_COND,      // Instruction executes conditionally
    Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
    // Load-store
    Constraint_WBSUPPRESS,
    Constraint_FAULT,
    Constraint_LIMITED_ATOMICITY, // Accesses are not single-copy atomic above the
    Constraint_NVNV1_00,
    Constraint_NVNV1_01,
    Constraint_NVNV1_11,
    Constraint_OSH,       // Constrain to Outer shareable
    Constraint_ISH,       // Constrain to Inner shareable
    Constraint_NSH,       // Constrain to Nonshareable

    Constraint_NC,        // Constrain to Noncacheable
    Constraint_WT,        // Constrain to Writethrough
    Constraint_WB,        // Constrain to Writeback

    // IPA too large
    Constraint_FORCE, Constraint_FORCENOSLCHECK,
    // PMSCR_PCT reserved values select Virtual timestamp
    Constraint_PMSCR_PCT_VIRT};
```



```

enumeration Unpredictable { // VMSR on MVFR
    Unpredictable_VMSR,
    // Writeback/transfer register overlap (load)
    Unpredictable_WBOVERLAPLD,
    // Writeback/transfer register overlap (store)
    Unpredictable_WBOVERLAPST,
    // Load Pair transfer register overlap
    Unpredictable_LDPOVERLAP,
    // Store-exclusive base/status register overlap
    Unpredictable_BASEOVERLAP,
    // Store-exclusive data/status register overlap
    Unpredictable_DATAOVERLAP,
    // Load-store alignment checks
    Unpredictable_DEVPAGE2,
    // Instruction fetch from Device memory
    Unpredictable_INSTRDEVICE,
    // Reserved CPACR value
    Unpredictable_RESCPACR,
    // Reserved MAIR value
    Unpredictable_RESMAIR,
    // Effect of SCTLR_ELx.C on Tagged attribute
    Unpredictable_S1CTAGGED,
    // Reserved Stage 2 MemAttr value
    Unpredictable_S2RESMEMATTR,
    // Reserved TEX:C:B value
    Unpredictable_RESTEXCB,
    // Reserved PRRR value
    Unpredictable_RESPPRR,
    // Reserved DACR field
    Unpredictable_RESDACR,
    // Reserved VTCR.S value
    Unpredictable_RESVTCRS,
    // Reserved TCR.TnSZ value
    Unpredictable_RESTnSZ,
    // Reserved SCTLR_ELx.TCF value
    Unpredictable_RESTCF,
    // Tag stored to Device memory
    Unpredictable_DEVICETAGSTORE,
    // Out-of-range TCR.TnSZ value
    Unpredictable_0ORTnSZ,
    // IPA size exceeds PA size
    Unpredictable_LARGEIPA,
    // Syndrome for a known-passing conditional A32 instruction
    Unpredictable_ESRCONDPASS,
    // Illegal State exception: zero PSTATE.IT
    Unpredictable_ILZEROIT,
    // Illegal State exception: zero PSTATE.T
    Unpredictable_ILZEROT,
    // Debug: prioritization of Vector Catch
    Unpredictable_BPVECTORCATCHPRI,
    // Debug Vector Catch: match on 2nd halfword
    Unpredictable_VCMATCHHALF,
    // Debug Vector Catch: match on Data Abort or Prefetch abort
    Unpredictable_VCMATCHDAPA,
    // Debug watchpoints: non-zero MASK and non-ones BAS
    Unpredictable_WPMASKANDBAS,
    // Debug watchpoints: non-contiguous BAS
    Unpredictable_WPBASCONTIGUOUS,
    // Debug watchpoints: reserved MASK
    Unpredictable_RESWPMASK,
    // Debug watchpoints: non-zero MASKed bits of address
    Unpredictable_WPMASKEDBITS,
    // Debug breakpoints and watchpoints: reserved control bits
    Unpredictable_RESBPWPCTRL,
    // Debug breakpoints: not implemented
    Unpredictable_BPNOTIMPL,
    // Debug breakpoints: reserved type
    Unpredictable_RESBPTYPE,
    // Debug breakpoints: not-context-aware breakpoint
    Unpredictable_BPNOTCTXCMP,

```

```

// Debug breakpoints: match on 2nd halfword of instruction
Unpredictable_BPMATCHHALF,
// Debug breakpoints: mismatch on 2nd halfword of instruction
Unpredictable_BPMISMATCHHALF,
// Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
// Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,
// Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,
// Zero top 32 bits of PC on illegal return to AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// FF speculation
Unpredictable_NONFAULT,
// Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,
// Load mem data in NF loads
Unpredictable_SVELDNFDATA,
// Write zeros in NF loads
Unpredictable_SVELDNFZERO,
// SP alignment fault when predicate is all zero
Unpredictable_CHECKSPNONEACTIVE,
// HCR_EL2.<NV,NV1> == '01'
Unpredictable_NVNV1,
// Reserved shareability encoding
Unpredictable_Shareability,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Consider SCTLR[].IESB in Debug state
Unpredictable_IESBinDebug,
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
Unpredictable_BADPMSFCR,
// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTTYPE,
// Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
// WFET or WFIT instruction in Debug state
Unpredictable_WFXTDEBUG,
// Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
// Misaligned exclusives, atomics, acquire/release to region that is not Normal
Unpredictable_MISALIGNEDATOMIC,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,
// ALUEXCEPTIONRETURN when in user/system mode in A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,
// Compare DBGxVR.RESS for BP/WP
Unpredictable_DBGxVR_RESS,
// Inaccessible event counter
Unpredictable_PMUEVENTCOUNTER,
// Reserved PMSCR.PCT behaviour.
Unpredictable_PMSCR_PCT};

```

Library pseudocode for shared/functions/vector/AdvSIMDEExpandImm

```
// AdvSIMDEExpandImm()
// =====

bits(64) AdvSIMDEExpandImm(bit op, bits(4) cmode, bits(8) imm8)
  case cmode<3:1> of
    when '000'
      imm64 = Replicate(Zeros(24):imm8, 2);
    when '001'
      imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
    when '010'
      imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
    when '011'
      imm64 = Replicate(imm8:Zeros(24), 2);
    when '100'
      imm64 = Replicate(Zeros(8):imm8, 4);
    when '101'
      imm64 = Replicate(imm8:Zeros(8), 4);
    when '110'
      if cmode<0> == '0' then
        imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
      else
        imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
    when '111'
      if cmode<0> == '0' && op == '0' then
        imm64 = Replicate(imm8, 8);
      if cmode<0> == '0' && op == '1' then
        imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
        imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
        imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
        imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
        imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
      if cmode<0> == '1' && op == '0' then
        imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5>:0>:Zeros(19);
        imm64 = Replicate(imm32, 2);
      if cmode<0> == '1' && op == '1' then
        if UsingAArch32() then ReservedEncoding();
        imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5>:0>:Zeros(48);

  return imm64;
```

Library pseudocode for shared/functions/vector/MatMulAdd

```
// MatMulAdd()
// =====
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned, boolean op2_unsigned)
  assert N == 128;

  bits(N) result;
  bits(32) sum;
  integer prod;

  for i = 0 to 1
    for j = 0 to 1
      sum = Elem[addend, 2*i + j, 32];
      for k = 0 to 7
        prod = Int(Elem[op1, 8*i + k, 8], op1_unsigned) * Int(Elem[op2, 8*j + k, 8], op2_unsigned)
        sum = sum + prod;
      Elem[result, 2*i + j, 32] = sum;

  return result;
```

Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// =====

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

Library pseudocode for shared/functions/vector/SatQ

```
// SatQ()
// =====

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```

Library pseudocode for shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// =====

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1; saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1)); saturated = TRUE;
    else
        result = i; saturated = FALSE;
    return (result<N-1:0>, saturated);
```

Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =====

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    if operand<N-1:N-2> == '00' then // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate = RecipSqrtEstimate(UInt(operand<31:23>), increasedprecision);
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =====

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N == 32;
    if operand<N-1> == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate = RecipEstimate(UInt(operand<31:23>), increasedprecision);

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

Library pseudocode for shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// =====

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1; saturated = TRUE;
    elsif i < 0 then
        result = 0; saturated = TRUE;
    else
        result = i; saturated = FALSE;
    return (result<N-1:0>, saturated);
```

Library pseudocode for shared/trace/selfhosted/SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// =====
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    if !HaveTraceExt() || !HaveSelfHostedTrace() then return FALSE;
    if HaveEL(EL3) then
        secure_trace_enable = (if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE);
        niden = (secure_trace_enable == '0' || ExternalSecureNoninvasiveDebugEnabled());
    else
        // If no EL3, IsSecure() returns the Effective value of (SCR_EL3.NS == '0')
        niden = (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled());
    return (EDSCR.TFO == '0' || !niden);
```

Library pseudocode for shared/trace/selfhosted/TraceAllowed

```
// TraceAllowed()
// =====
// Returns TRUE if Self-hosted Trace is allowed in the current Security state and Exception level

boolean TraceAllowed()
    if !HaveTraceExt() then return FALSE;
    if SelfHostedTraceEnabled() then
        if IsSecure() && HaveEL(EL3) then
            secure_trace_enable = (if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE);
            if secure_trace_enable == '0' then return FALSE;
        TGE_bit = if EL2Enabled() then HCR_EL2.TGE else '0';
        case PSTATE.EL of
            when EL3 TRE_bit = if !TRE_bit = if HaveAArch64HighestELUsingAArch32() then TRFCR.E1TRE else
            when EL2 TRE_bit = TRFCR_EL2.E2TRE;
            when EL1 TRE_bit = TRFCR_EL1.E1TRE;
            when EL0 TRE_bit = if TGE_bit == '1' then TRFCR_EL2.E0HTRE else TRFCR_EL1.E0TRE;
        return TRE_bit == '1';
    else
        return (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled());
```

Library pseudocode for shared/trace/selfhosted/TraceContextIDR2

```
// TraceContextIDR2()
// =====

boolean TraceContextIDR2()
    if !TraceAllowed() || !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');
```

Library pseudocode for shared/trace/selfhosted/TraceSynchronizationBarrier

```
// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers
TraceSynchronizationBarrier();
```


Library pseudocode for shared/trace/selfhosted/TraceTimeStamp

```
// TraceTimeStamp()
// =====

TimeStamp TraceTimeStamp()
  if SelfHostedTraceEnabled() then
    if HaveEL(EL2) then
      TS_el2 = TRFCR_EL2.TS;
      if !HaveECVExt() && TS_el2 == '10' then
        // Reserved value
        (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP);

      case TS_el2 of
        when '00'
          // Falls out to check TRFCR_EL1.TS
        when '01'
          return TimeStamp_Virtual;
        when '10'
          assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
          return TimeStamp_OffsetPhysical;
        when '11'
          return TimeStamp_Physical;

      TS_el1 = TRFCR_EL1.TS;
      if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
        // Reserved value
        (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP);

      case TS_el1 of
        when '01'
          return TimeStamp_Virtual;
        when '10'
          assert HaveECVExt();
          return TimeStamp_OffsetPhysical;
        when '11'
          return TimeStamp_Physical;
        otherwise
          Unreachable(); // ConstrainUnpredictableBits removes this case
    else
      return TimeStamp_CoreSight;
```

Library pseudocode for shared/translation/attrs/DecodeDevice

```
// DecodeDevice()
// =====
// Decode output Device type

DeviceType DecodeDevice(bits(2) device)
  case device of
    when '00' return DeviceType_nGnRnE;
    when '01' return DeviceType_nGnRE;
    when '10' return DeviceType_nGRE;
    when '11' return DeviceType_GRE;
```

Library pseudocode for shared/translation/attrs/DecodeLDFAttr

```
// DecodeLDFAttr()
// =====
// Decode memory attributes using LDF (Long Descriptor Format) mapping

MemAttrHints DecodeLDFAttr(bits(4) attr)
    MemAttrHints ldfattr;

    if attr == 'x0xx' then ldfattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldfattr.attrs = MemAttr_NC; // Non-cacheable
    elsif attr == 'x1xx' then ldfattr.attrs = MemAttr_WB; // Write-back
    else
        Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint_No; // No allocation hints
            when '01' ldfattr.hints = MemHint_WA; // Write-allocate
            when '10' ldfattr.hints = MemHint_RA; // Read-allocate
            when '11' ldfattr.hints = MemHint_RWA; // Read/Write allocate

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldfattr.attrs != MemAttr_NC && ldfattr.hints != MemHint_No then
        ldfattr.transient = attr<3> == '0';

    return ldfattr;
```

Library pseudocode for shared/translation/attrs/DecodeSDFAttr

```
// DecodeSDFAttr()
// =====
// Decode memory attributes using SDF (Short Descriptor Format) mapping

MemAttrHints DecodeSDFAttr(bits(2) rgn)
    MemAttrHints sdfattr;

    case rgn of
        when '00' // Non-cacheable (no allocate)
            sdfattr.attrs = MemAttr_NC;
        when '01' // Write-back, Read and Write allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RWA;
        when '10' // Write-through, Read allocate
            sdfattr.attrs = MemAttr_WT;
            sdfattr.hints = MemHint_RA;
        when '11' // Write-back, Read allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RA;

    sdfattr.transient = FALSE;

    return sdfattr;
```

Library pseudocode for shared/translation/attrs/DecodeShareability

```
// DecodeShareability()
// =====
// Decode shareability of target memory region

Shareability DecodeShareability(bits(2) sh)
    case sh of
        when '10' return Shareability\_OSH;
        when '11' return Shareability\_ISH;
        when '00' return Shareability\_NSH;
        otherwise
            case ConstrainUnpredictable\(Unpredictable\_Shareability\) of
                when Constraint\_OSH return Shareability\_OSH;
                when Constraint\_ISH return Shareability\_ISH;
                when Constraint\_NSH return Shareability\_NSH;
```

Library pseudocode for shared/translation/attrs/MAIRAttr

```
// MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) MAIRAttr(integer index, MAIRType mair)
    bit_index = 8 * index;
    return mair<bit_index+7:bit_index>;
```

Library pseudocode for shared/translation/attrs/NormalNCMemAttr

```
// NormalNCMemAttr()
// =====
// Normal Non-cacheable memory attributes

MemoryAttributes NormalNCMemAttr()
    MemAttrHints non_cacheable;
    non_cacheable.attrs = MemAttr\_NC;

    MemoryAttributes nc_memattrs;
    nc_memattrs.memtype      = MemType\_Normal;
    nc_memattrs.outer        = non_cacheable;
    nc_memattrs.inner        = non_cacheable;
    nc_memattrs.shareability = Shareability\_OSH;
    nc_memattrs.tagged        = FALSE;

    return nc_memattrs;
```

Library pseudocode for shared/translation/attrs/NormaliseShareability

```
// NormaliseShareability()
// =====
// Force Outer Shareability on Device and Normal iNCoNC memory

Shareability NormaliseShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType\_Device ||
        (memattrs.inner.attrs == MemAttr\_NC &&
         memattrs.outer.attrs == MemAttr\_NC)) then
        return Shareability\_OSH;
    else
        return memattrs.shareability;
```

Library pseudocode for shared/translation/attrs/S1ConstrainUnpredictableRESMAIR

```
// S1ConstrainUnpredictableRESMAIR()
// =====
// Determine whether a reserved value occupies MAIR_ELx.AttrN

boolean S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean slaarch64)
  case attr of
    when '0000xx01' return !(slaarch64 && HaveFeatXS\(\));
    when '0000xxxx' return attr<1:0> != '00';
    when '01000000' return !(slaarch64 && HaveFeatXS\(\));
    when '10100000' return !(slaarch64 && HaveFeatXS\(\));
    when '11110000' return !(slaarch64 && HaveMTE2Ext\(\));
    when 'xxxx0000' return TRUE;
    otherwise       return FALSE;
```



```

// S1DecodeMemAttrs()
// =====
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr, bits(2) sh, boolean slaarch64)
  if S1ConstrainUnpredictableRESMAIR(attr, slaarch64) then
    (-, attr) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);

  MemoryAttributes memattrs;
  case attr of
    when '0000xxxx' // Device memory
      memattrs.memtype = MemType\_Device;
      memattrs.device = DecodeDevice(attr<3:2>);
      memattrs.tagged = FALSE;
      memattrs.xs = if slaarch64 then NOT attr<0> else '1';
    when '01000000'
      assert slaarch64 && memattrs.shareability = DecodeShareability(sh);
      memattrs.tagged = FALSE;
      memattrs.xs = if slaarch64 then NOT attr<0> else '1';
    when '01000000'
      assert slaarch64 && HaveFeatXS();
      memattrs.memtype = MemType\_Normal;
      memattrs.tagged = FALSE;
      memattrs.outer.attrs = MemAttr\_NC;
      memattrs.inner.attrs = MemAttr\_NC;
      memattrs.xs = '0';
    when '10100000'
      assert slaarch64 && memattrs.shareability = DecodeShareability(sh);
      memattrs.xs = '0';
    when '10100000'
      assert slaarch64 && HaveFeatXS();
      memattrs.memtype = MemType\_Normal;
      memattrs.tagged = FALSE;
      memattrs.outer.attrs = memattrs.tagged = FALSE;
      memattrs.shareability = DecodeShareability(sh);
      memattrs.outer.attrs = MemAttr\_WT;
      memattrs.outer.hints = MemHint\_RA;
      memattrs.outer.transient = FALSE;
      memattrs.inner.attrs = MemAttr\_WT;
      memattrs.inner.hints = MemHint\_RA;
      memattrs.inner.transient = FALSE;
      memattrs.xs = '0';
    when '11110000' // Tagged memory
      assert slaarch64 && HaveMTE2Ext();
      memattrs.memtype = MemType\_Normal;
      memattrs.tagged = TRUE;
      memattrs.outer.attrs = memattrs.tagged = TRUE;
      memattrs.shareability = DecodeShareability(sh);
      memattrs.outer.attrs = MemAttr\_WB;
      memattrs.outer.hints = MemHint\_RWA;
      memattrs.outer.transient = FALSE;
      memattrs.inner.attrs = MemAttr\_WB;
      memattrs.inner.hints = MemHint\_RWA;
      memattrs.inner.transient = FALSE;
      memattrs.xs = '0';
    otherwise
      memattrs.memtype = MemType\_Normal;
      memattrs.outer = DecodeLDFAttr(attr<7:4>);
      memattrs.inner = DecodeLDFAttr(attr<3:0>);
      memattrs.tagged = FALSE;
      if (memattrs.inner.attrs == memattrs.shareability = DecodeShareability(sh);
      memattrs.tagged = FALSE;
      if (memattrs.inner.attrs == MemAttr\_WB &&
          memattrs.outer.attrs == MemAttr\_WB) then
        memattrs.xs = '0';
      else

```

```

        memattrs.xs = '1';
    memattrs.shareability = DecodeShareability(sh);
) then
    memattrs.xs = '0';
else
    memattrs.xs = '1';

return memattrs;

```

Library pseudocode for shared/translation/attrs/S2CombineS1AttrHints

```

// S2CombineS1AttrHints()
// =====
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

MemAttrHints S2CombineS1AttrHints(MemAttrHints s1_attrhints, MemAttrHints s2_attrhints)
    MemAttrHints attrhints;

    if s1_attrhints.attrs == MemAttr_NC || s2_attrhints.attrs == MemAttr_NC then
        attrhints.attrs = MemAttr_NC;
    elsif s1_attrhints.attrs == MemAttr_WT || s2_attrhints.attrs == MemAttr_WT then
        attrhints.attrs = MemAttr_WT;
    else
        attrhints.attrs = MemAttr_WB;

    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != MemAttr_NC then
        attrhints.hints = s1_attrhints.hints;
        attrhints.transient = s1_attrhints.transient;

    return attrhints;

```

Library pseudocode for shared/translation/attrs/S2CombineS1Device

```

// S2CombineS1Device()
// =====
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

DeviceType S2CombineS1Device(DeviceType s1_device, DeviceType s2_device)
    if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE then
        return DeviceType_nGnRnE;
    elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
        return DeviceType_nGnRE;
    elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
        return DeviceType_nGRE;
    else
        return DeviceType_GRE;

```

Library pseudocode for shared/translation/attrs/S2CombineS1MemAttrs

```
// S2CombineS1MemAttrs()
// =====
// Combine stage 2 with stage 1 memory attributes

MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_memattrs,
                                     MemoryAttributes s2_memattrs)
    MemoryAttributes memattrs;

    if s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype == MemType_Device then
        memattrs.memtype = MemType_Device;
        memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
    elseif s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype == MemType_Device then
        memattrs.shareability = Shareability_OSH;
    elseif s1_memattrs.memtype == MemType_Device then // S2 Normal, S1 Device
        memattrs = s1_memattrs;
    elseif s2_memattrs.memtype == MemType_Device && s1_memattrs.memtype == MemType_Device then
        memattrs.shareability = Shareability_OSH;
    elseif s2_memattrs.memtype == MemType_Device then // S2 Device, S1 Normal
        memattrs = s2_memattrs;
    else // S2 Normal, S1 Normal
        memattrs.memtype = MemType_Normal;
        memattrs.shareability = Shareability_OSH;
    else // S2 Normal, S1 Normal
        memattrs.memtype = MemType_Normal;
        memattrs.inner = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
        memattrs.outer = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);

    if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
        memattrs.shareability = Shareability_OSH;
    else
        memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                         s2_memattrs.shareability);

    if ELUsingAArch32(EL2) || !HaveMTE2Ext() then
        memattrs.tagged = FALSE;
    else
        memattrs.tagged = AArch64.IsS2ResultTagged(memattrs, s1_memattrs.tagged);

    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_memattrs.shareability);
    memattrs.xs = s2_memattrs.xs;

    memattrs.shareability = NormaliseShareability(memattrs);
    (memattrs, s1_memattrs.tagged);

    memattrs.xs = s2_memattrs.xs;

    return memattrs;
```

Library pseudocode for shared/translation/attrs/S2CombineS1Shareability

```
// S2CombineS1Shareability()
// =====
// Combine stage 2 shareability with stage 1

Shareability S2CombineS1Shareability(Shareability s1_shareability,
                                     Shareability s2_shareability)

    if (s1_shareability == Shareability_OSH ||
        s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elseif (s1_shareability == Shareability_ISH ||
            s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
```


Library pseudocode for shared/translation/attrs/S2DecodeCacheability

```
// S2DecodeCacheability()
// =====
// Determine the stage 2 cacheability for Normal memory

MemAttrHints S2DecodeCacheability(bits(2) attr)
    MemAttrHints s2attr;

    case attr of
        when '01' s2attr.attrs = MemAttr_NC; // Non-cacheable
        when '10' s2attr.attrs = MemAttr_WT; // Write-through
        when '11' s2attr.attrs = MemAttr_WB; // Write-back
        otherwise // Constrained unpredictable
            case ConstrainUnpredictable(Unpredictable_S2RESMEMATTR) of
                when Constraint_NC s2attr.attrs = MemAttr_NC;
                when Constraint_WT s2attr.attrs = MemAttr_WT;
                when Constraint_WB s2attr.attrs = MemAttr_WB;

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints = bits(2) UNKNOWN;
    s2attr.transient = boolean UNKNOWN;

    return s2attr;
```

Library pseudocode for shared/translation/attrs/S2DecodeMemAttrs

```
// S2DecodeMemAttrs()
// =====
// Decode stage 2 memory attributes

MemoryAttributes S2DecodeMemAttrs(bits(4) attr, bits(2) sh)
    MemoryAttributes memattrs;

    case attr of
        when '00xx' // Device memory
            memattrs.memtype = MemType_Device;
            memattrs.device = DecodeDevice(attr<1:0>);
        otherwise // Normal memory
            memattrs.memtype = memattrs.shareability = Shareability_OSH;
        otherwise // Normal memory
            memattrs.memtype = MemType_Normal;
            memattrs.outer = S2DecodeCacheability(attr<3:2>);
            memattrs.inner = S2DecodeCacheability(attr<1:0>);

            if (memattrs.inner.attrs == MemAttr_NC &&
                memattrs.outer.attrs == MemAttr_NC) then
                memattrs.shareability = Shareability_OSH(attr<1:0>);

            memattrs.shareability =;
        else
            memattrs.shareability = DecodeShareability(sh);

    return memattrs;
```

Library pseudocode for shared/translation/attrs/WalkMemAttrs

```
// WalkMemAttrs()
// =====
// Retrieve memory attributes of translation table walk

MemoryAttributes WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn)
    MemoryAttributes walkmemattrs;

    walkmemattrs.memtype      = MemType\_Normal;
    walkmemattrs.shareability = DecodeShareability(sh);
    walkmemattrs.inner       = DecodeSDFAttr(irgn);
    walkmemattrs.outer       = DecodeSDFAttr(orgn);
    walkmemattrs.tagged      = FALSE;
    if (walkmemattrs.inner.attrs == MemAttr\_WB &&
        walkmemattrs.outer.attrs == MemAttr\_WB) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';

    return walkmemattrs;
```

Library pseudocode for shared/translation/faults/AlignmentFault

```
// AlignmentFault()
// =====

FaultRecord AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)
    FaultRecord fault;

    fault.statuscode = Fault\_Alignment;
    fault.acctype    = acctype;
    fault.write      = iswrite;
    fault.secondstage = secondstage;

    return fault;
```

Library pseudocode for shared/translation/faults/AsyncExternalAbort

```
// AsyncExternalAbort()
// =====
// Return a fault record indicating an asynchronous external abort

FaultRecord AsyncExternalAbort(boolean parity, bits(2) errortype, bit extflag)
    FaultRecord fault;

    fault.statuscode = if parity then Fault\_AsyncParity else Fault\_AsyncExternal;
    fault.extflag    = extflag;
    fault.errortype   = errortype;
    fault.acctype     = AccType\_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fslwalk   = FALSE;

    return fault;
```

Library pseudocode for shared/translation/faults/NoFault

```
// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred

FaultRecord NoFault()
    FaultRecord fault;

    fault.statuscode = Fault_None;
    fault.acctype    = AccType_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fslwalk  = FALSE;

    return fault;
```

Library pseudocode for shared/translation/translation/HasS2Translation

```
// HasS2Translation()
// =====
// Returns TRUE if stage 2 translation is present for the current translation regime

boolean HasS2Translation()
    return (EL2Enabled() && !IsInHost() && PSTATE.EL IN {EL0, EL1});
```

Library pseudocode for shared/translation/translation/Have16bitVMID

```
// Have16bitVMID()
// =====
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return HaveEL(EL2) && boolean IMPLEMENTATION_DEFINED "Has 16-bit VMID";
```

Library pseudocode for shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

Library pseudocode for shared/translation/vmsa/CreateAddressDescriptor

```
// CreateAddressDescriptor()
// =====
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(bits(64) va, FullAddress pa,
                                         MemoryAttributes memattrs)
    AddressDescriptor addrdesc;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault();

    return addrdesc;
```

Library pseudocode for shared/translation/vmsa/CreateFaultyAddressDescriptor

```
// CreateFaultyAddressDescriptor()
// =====
// Set internal members for address descriptor type with values indicating error

AddressDescriptor CreateFaultyAddressDescriptor(bits(64) va, FaultRecord fault)
    AddressDescriptor addrdesc;

    addrdesc.vaddress = va;
    addrdesc.fault    = fault;

    return addrdesc;
```

Library pseudocode for shared/translation/vmsa/DescriptorType

```
enumeration DescriptorType {
    DescriptorType_Table,
    DescriptorType_Block,
    DescriptorType_Page,
    DescriptorType_Invalid
};
```

Library pseudocode for shared/translation/vmsa/Domains

```
constant bits(2) Domain_NoAccess = '00';
constant bits(2) Domain_Client   = '01';
constant bits(2) Domain_Manager  = '11';
```

Library pseudocode for shared/translation/vmsa/FetchDescriptor

```
// FetchDescriptor()
// =====
// Fetch a translation table descriptor

(FaultRecord, bits(N))(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress,
    FaultRecord fault)
    // 32-bit descriptors for AArch32 Short-descriptor format
    // 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
    assert N == 32 || N == 64;
    bits(N) descriptor;

    // These parameters are for the Abort() call on the other side of _Mem
    walkacc = FetchDescriptor(bit ee, AddressDescriptor walkaddress,
        FaultRecord fault)
    // 32-bit descriptors for AArch32 Short-descriptor format
    // 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
    assert N == 32 || N == 64;
    bits(N) descriptor;

    walkacc = CreateAccessDescriptor(AccType_TTW);
    (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkacc);
    if IsFault(memstatus) then
        fault = HandleExternalTTWAbort(memstatus, fault.write, walkaddress,
            walkacc, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);
(fault.acctype, fault.secondstage,
    fault.s2fslwalk, fault.level);

    descriptor = _Mem[walkaddress, N DIV 8, walkacc, fault.write];

    if ee == '1' then
        descriptor = BigEndianReverse(descriptor);

    return (fault, descriptor);
```

Library pseudocode for shared/translation/vmsa/HasUnprivileged

```
// HasUnprivileged()
// =====
// Returns whether a translation regime serves EL0 as well as a higher EL

boolean HasUnprivileged(Regime regime)
    return (regime IN {
        Regime_EL20,
        Regime_EL30,
        Regime_EL10
    });
```

Library pseudocode for shared/translation/vmsa/IsAtomicRW

```
// IsAtomicRW()
// =====
// Is the access an atomic operation?

boolean IsAtomicRW(AccType acctype)
    return acctype IN {
        AccType_ATOMICRW,
        AccType_ORDEREDRW,
        AccType_ORDEREDATOMICRW
    };
```

Library pseudocode for shared/translation/vmsa/Regime

```
enumeration Regime {
    Regime_EL3,           // EL3
    Regime_EL30,          // EL3&0 (PL1&0 when EL3 is AArch32)
    Regime_EL2,           // EL2
    Regime_EL20,          // EL2&0
    Regime_EL10           // EL1&0
};
```

Library pseudocode for shared/translation/vmsa/RegimeUsingAArch32

```
// RegimeUsingAArch32()
// =====
// Determine if the EL controlling the regime executes in AArch32 state

boolean RegimeUsingAArch32(Regime regime)
    case regime of
        when Regime_EL10 return ELUsingAArch32(EL1);
        when Regime_EL30 return TRUE;
        when Regime_EL20 return FALSE;
        when Regime_EL2 return ELUsingAArch32(EL2);
        when Regime_EL3 return FALSE;
```

Library pseudocode for shared/translation/vmsa/S1TTWParams

```
type S1TTWParams is (
// A64-VMSA exclusive parameters
    bit      ha,          // TCR_ELx.HA
    bit      hd,          // TCR_ELx.HD
    bit      tbi,         // TCR_ELx.TBI{x}
    bit      tbid,        // TCR_ELx.TBID{x}
    bit      nfd,         // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EL2.E2H == '1'
    bit      e0pd,        // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bit      ds,          // TCR_ELx.DS
    bits(3)  ps,          // TCR_ELx.{I}PS
    bits(6)  txsz,        // TCR_ELx.TxSZ
    bit      epan,        // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    bit      dct,         // HCR_EL2.DCT
    bit      nv1,         // HCR_EL2.NV1

// A32-VMSA exclusive parameters
    bits(3)  t0sz,        // TTBCR.T0SZ
    bits(3)  t1sz,        // TTBCR.T1SZ
    bit      uwxn,        // SCTLR.UWXN

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    TGx      tgx,         // TCR_ELx.TGx      / Always TGx_4KB
    bits(2)  irgn,        // TCR_ELx.IRGNx   / TTBCR.IRGNx or HTCR.IRGN0
    bits(2)  orgn,        // TCR_ELx.ORGNx   / TTBCR.ORGX or HTCR.ORGX0
    bits(2)  sh,          // TCR_ELx.SHx     / TTBCR.SHx or HTCR.SH0
    bit      hpd,         // TCR_ELx.HPD{x}  / TTBCR2.HPDx or HTCR.HPD
    bit      ee,          // SCTLR_ELx.EE     / SCTLR.EE or HSCTLR.EE
    bit      wxn,         // SCTLR_ELx.WXN    / SCTLR.WXN or HSCTLR.WXN
    bit      ntlsm,       // SCTLR_ELx.nTlSMD / SCTLR.nTlSMD or HSCTLR.nTlSMD
    bit      dc,          // HCR_EL2.DC       / HCR.DC
    bit      sif,         // SCR_EL3.SIF      / SCR.SIF
    MAIRType mair         // MAIR_ELx         / MAIR1:MAIR0 or HMAIR1:HMAIR0
);
```

Library pseudocode for shared/translation/vmsa/S2TTWParams

```
type S2TTWParams is (  
  // A64-VMSA exclusive parameters  
  bit      ha,      // VTCR_EL2.HA  
  bit      hd,      // VTCR_EL2.HD  
  bit      sl2,     // V{S}TCR_EL2.SL2  
  bit      ds,      // VTCR_EL2.DS  
  bit      sw,      // VSTCR_EL2.SW  
  bit      nsw,     // VTCR_EL2.NSW  
  bit      sa,      // VSTCR_EL2.SA  
  bit      nsa,     // VTCR_EL2.NSA  
  bits(3)  ps,      // VTCR_EL2.PS  
  bits(6)  txsz,    // V{S}TCR_EL2.T0SZ  
  bit      fwb,     // HCR_EL2.PTW  
  
  // A32-VMSA exclusive parameters  
  bit      s,       // VTCR.S  
  bits(4)  t0sz,    // VTCR.T0SZ  
  
  // Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)  
  TGx      tgx,     // V{S}TCR_EL2.TG0 / Always TGx_4KB  
  bits(2)  sl0,     // V{S}TCR_EL2.SL0 / VTCR.SL0  
  bits(2)  irgn,    // VTCR_EL2.IRGN0 / VTCR.IRGN0  
  bits(2)  orgn,    // VTCR_EL2.ORGNO / VTCR.ORGNO  
  bits(2)  sh,      // VTCR_EL2.SH0 / VTCR.SH0  
  bit      ee,      // SCTLR_EL2.EE / HSCTLR.EE  
  bit      ptw,     // HCR_EL2.PTW / HCR.PTW  
  bit      vm,      // HCR_EL2.VM / HCR.VM  
)  
  
constant integer FINAL_LEVEL = 3;
```

Library pseudocode for shared/translation/vmsa/SDFTType

```
enumeration SDFTType {  
  SDFTType_Table,  
  SDFTType_Invalid,  
  SDFTType_Supersection,  
  SDFTType_Section,  
  SDFTType_LargePage,  
  SDFTType_SmallPage  
};
```

Library pseudocode for shared/translation/vmsa/SecurityStateForRegime

```
// SecurityStateForRegime()  
// =====  
// Return the Security State of the given translation regime  
  
SecurityState SecurityStateForRegime(Regime regime)  
  case regime of  
    when Regime_EL3      return SecurityStateAtEL(EL3);  
    when Regime_EL30     return SS_Secure; // A32 EL3 is always Secure  
    when Regime_EL2      return SecurityStateAtEL(EL2);  
    when Regime_EL20     return SecurityStateAtEL(EL2);  
    when Regime_EL10     return SecurityStateAtEL(EL1);
```

Library pseudocode for shared/translation/vmsa/StageOA

```
// StageOA()
// =====
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

FullAddress StageOA(FullAddress outputbase, bits(64) ia, TGx tgx, integer level)
    // Output Address
    FullAddress oa;

    tsize = TranslationSize(tgx, level);

    oa.paspace = outputbase.paspace;
    oa.address = outputbase.address<51:tsize>:ia<tsize-1:0>;

    return oa;
```

Library pseudocode for shared/translation/vmsa/TGx

```
enumeration TGx {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
```

Library pseudocode for shared/translation/vmsa/TGxGranuleBits

```
// TGxGranuleBits()
// =====
// Retrieve the address size, in bits, of a granule

integer TGxGranuleBits(TGx tgx)
    case tgx of
        when TGx_4KB return 12;
        when TGx_16KB return 14;
        when TGx_64KB return 16;
```

Library pseudocode for shared/translation/vmsa/TTWState

```
type TTWState is (
    boolean          istable,
    integer          level,
    FullAddress      baseaddress,
    bit             contiguous,
    bit             guardedpage,
    SDFType         sdftype, // AArch32 Short-descriptor format walk only
    bits(4)         domain, // AArch32 Short-descriptor format walk only
    MemoryAttributes memattrs,
    Permissions      permissions
)
```


Library pseudocode for shared/translation/vmsa/TranslationRegime

```
// TranslationRegime()
// =====
// Select the translation regime given the target EL and PE state

Regime TranslationRegime(bits(2) el, AccType acctype)
    if el == EL3 then
        return if ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL3;
    elsif el == EL2 then
        return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
    elsif el == EL1 then
        if acctype == AccType_NV2REGISTER then
            assert EL2Enabled();
            return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
        else
            return Regime_EL10;
    elsif el == EL0 then
        if IsSecure() && ELUsingAArch32(EL3) then
            return Regime_EL30;
        elsif ELIsInHost(EL0) then
            return Regime_EL20;
        else
            return Regime_EL10;
    else
        Unreachable();
```

Library pseudocode for shared/translation/vmsa/TranslationSize

```
// TranslationSize()
// =====
// Compute the number of bits directly mapped from the input address
// to the output address

integer TranslationSize(TGx tgx, integer level)
    granulebits = TGxGranuleBits(tgx);
    blockbits = (FINAL_LEVEL - level) * (granulebits - 3);

    return granulebits + blockbits;
```

Library pseudocode for shared/translation/vmsa/VARange

```
enumeration VARange {
    VARange_LOWER,
    VARange_UPPER
};
```

Internal version only: isa v32.21v32.15, AdvSIMD v29.05, pseudocode v2021-06_xmlv2021-03, sve v2021-06_rc2bv2021-03_rc2 ; Build timestamp: 2021-06-28T16:20:41Z2021-03-30T21:41:22

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)