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 Limited ("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 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, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement specifically covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the ARM trademark usage guidelines http://www.arm.com/about/trademarks/guidelines/index.php.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20327

In this document, where the term ARM is used to refer to the company it means "ARM or any of its subsidiaries as appropriate".

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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).

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.

B: Branch.

B.cond: Branch conditionally.

BFC: Bitfield Clear, leaving other bits unchanged: 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.

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).

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 zero bits.

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

CMN (immediate): Compare Negative (immediate): an alias of ADDS (immediate).

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

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

CMP (immediate): Compare (immediate): an alias of SUBS (immediate).

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

CNEG: Conditional Negate: an alias of CSNEG.

CRC32B, CRC32H, CRC32W, CRC32X: CRC32 checksum.

CRC32CB, CRC32CH, CRC32CW, CRC32CX: CRC32C checksum.

CSEL: Conditional Select.

CSET: Conditional Set: an alias of CSINC.

CSETM: Conditional Set Mask: an alias of CSINV.

CSINC: Conditional Select Increment.

CSINV: Conditional Select Invert.

CSNEG: Conditional Select Negation.

DC: Data Cache operation: an alias of SYS.

DCPS1: Debug Change PE State to EL1..

DCPS2: Debug Change PE State to EL2..

DCPS3: Debug Change PE State to EL3.

DMB: Data Memory Barrier.

DRPS: Debug restore process state.

DSB: Data Synchronization Barrier.

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.

HINT: Hint instruction.

HLT: Halt instruction.

HVC: Hypervisor Call.

IC: Instruction Cache operation: an alias of SYS.

ISB: Instruction Synchronization Barrier.

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

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

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

LDAPR: Load-Acquire RCpc Register.

LDAPRB: Load-Acquire RCpc Register Byte.

LDAPRH: Load-Acquire RCpc Register Halfword.

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.

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).

PRFM (unscaled offset): Prefetch Memory (unscaled offset).

PSB CSYNC: Profiling Synchronization Barrier.

RBIT: Reverse Bits.

RET: Return from subroutine.

RETAA, RETAB: Return from subroutine, with pointer authentication.

REV: Reverse Bytes.

REV16: Reverse bytes in 16-bit halfwords.

REV32: Reverse bytes in 32-bit words.

REV64: Reverse Bytes: an alias of REV.

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

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

RORV: Rotate Right Variable.

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.

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.

STADD, STADDL: Atomic add on word or doubleword in memory, without return.

STADDB, STADDLB: Atomic add on byte in memory, without return.

STADDH, STADDLH: Atomic add on halfword in memory, without return.

STCLR, STCLRL: Atomic bit clear on word or doubleword in memory, without return.

STCLRB, STCLRLB: Atomic bit clear on byte in memory, without return.

STCLRH, STCLRLH: Atomic bit clear on halfword in memory, without return.

STEOR, STEORL: Atomic exclusive OR on word or doubleword in memory, without return.

STEORB, STEORLB: Atomic exclusive OR on byte in memory, without return.

STEORH, STEORLH: Atomic exclusive OR on halfword in memory, without return.

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.

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.

STSETB, STSETLB: Atomic bit set on byte in memory, without return.

STSETH, STSETLH: Atomic bit set on halfword in memory, without return.

STSMAX, STSMAXL: Atomic signed maximum on word or doubleword in memory, without return.

STSMAXB, STSMAXLB: Atomic signed maximum on byte in memory, without return.

STSMAXH, STSMAXLH: Atomic signed maximum on halfword in memory, without return.

STSMIN, STSMINL: Atomic signed minimum on word or doubleword in memory, without return.

STSMINB, STSMINLB: Atomic signed minimum on byte in memory, without return.

STSMINH, STSMINLH: Atomic signed minimum on halfword in memory, without return.

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.

STUMAXB, STUMAXLB: Atomic unsigned maximum on byte in memory, without return.

STUMAXH, STUMAXLH: Atomic unsigned maximum on halfword in memory, without return.

STUMIN, STUMINL: Atomic unsigned minimum on word or doubleword in memory, without return.

STUMINB, STUMINLB: Atomic unsigned minimum on byte in memory, without return.

STUMINH, STUMINLH: Atomic unsigned minimum on halfword in memory, without return.

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.

SUB (extended register): Subtract (extended register).

SUB (immediate): Subtract (immediate).

SUB (shifted register): Subtract (shifted register).

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.

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.

UDIV: Unsigned Divide.

UMADDL: Unsigned Multiply-Add Long.

UMNEGL: Unsigned Multiply-Negate Long: an alias of UMSUBL.

UMSUBL: Unsigned Multiply-Subtract Long.

UMULH: Unsigned Multiply High.

UMULL: Unsigned Multiply Long: an alias of UMADDL.

UXTB: Unsigned Extend Byte: an alias of UBFM.

UXTH: Unsigned Extend Halfword: an alias of UBFM.

WFE: Wait For Event.

WFI: Wait For Interrupt.

XPACD, XPACI, XPACLRI: Strip Pointer Authentication Code.

YIELD: YIELD.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

ABS: Absolute value (vector).

ADD (vector): Add (vector).

ADDHN, ADDHN2: Add returning High Narrow.

ADDP (scalar): Add Pair of elements (scalar).

ADDP (vector): Add Pairwise (vector).

ADDV: Add across Vector.

AESD: AES single round decryption.

AESE: AES single round encryption.

AESIMC: AES inverse mix columns.

AESMC: AES mix columns.

AND (vector): Bitwise AND (vector).

BCAX: Bit Clear and XOR.

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

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

BIF: Bitwise Insert if False.

BIT: Bitwise Insert if True.

BSL: Bitwise Select.

CLS (vector): Count Leading Sign bits (vector).

CLZ (vector): Count Leading Zero bits (vector).

CMEQ (register): Compare bitwise Equal (vector).

CMEQ (zero): Compare bitwise Equal to zero (vector).

CMGE (register): Compare signed Greater than or Equal (vector).

CMGE (zero): Compare signed Greater than or Equal to zero (vector).

CMGT (register): Compare signed Greater than (vector).

CMGT (zero): Compare signed Greater than zero (vector).

CMHI (register): Compare unsigned Higher (vector).

CMHS (register): Compare unsigned Higher or Same (vector).

CMLE (zero): Compare signed Less than or Equal to zero (vector).

CMLT (zero): Compare signed Less than zero (vector).

CMTST: Compare bitwise Test bits nonzero (vector).

CNT: Population Count per byte.

DUP (element): Duplicate vector element to vector or scalar.

DUP (general): Duplicate general-purpose register to vector.

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

EOR3: Three-way Exclusive OR.

EXT: Extract vector from pair of vectors.

FABD: Floating-point Absolute Difference (vector).

FABS (scalar): Floating-point Absolute value (scalar).

FABS (vector): Floating-point Absolute value (vector).

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

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

FADD (scalar): Floating-point Add (scalar).

FADD (vector): Floating-point Add (vector).

FADDP (scalar): Floating-point Add Pair of elements (scalar).

FADDP (vector): Floating-point Add Pairwise (vector).

FCADD: Floating-point Complex Add.

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

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

FCMEQ (register): Floating-point Compare Equal (vector).

FCMEQ (zero): Floating-point Compare Equal to zero (vector).

FCMGE (register): Floating-point Compare Greater than or Equal (vector).

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

FCMGT (register): Floating-point Compare Greater than (vector).

FCMGT (zero): Floating-point Compare Greater than zero (vector).

FCMLA: Floating-point Complex Multiply Accumulate.

FCMLA (by element): Floating-point Complex Multiply Accumulate (by element).

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

FCMLT (zero): Floating-point Compare Less than zero (vector).

FCMP: Floating-point quiet Compare (scalar).

FCMPE: Floating-point signaling Compare (scalar).

FCSEL: Floating-point Conditional Select (scalar).

FCVT: Floating-point Convert precision (scalar).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

FDIV (scalar): Floating-point Divide (scalar).

FDIV (vector): Floating-point Divide (vector).

FJCVTZS: Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.

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

FMAX (scalar): Floating-point Maximum (scalar).

FMAX (vector): Floating-point Maximum (vector).

FMAXNM (scalar): Floating-point Maximum Number (scalar).

FMAXNM (vector): Floating-point Maximum Number (vector).

FMAXNMP (scalar): Floating-point Maximum Number of Pair of elements (scalar).

FMAXNMP (vector): Floating-point Maximum Number Pairwise (vector).

FMAXNMV: Floating-point Maximum Number across Vector.

FMAXP (scalar): Floating-point Maximum of Pair of elements (scalar).

FMAXP (vector): Floating-point Maximum Pairwise (vector).

FMAXV: Floating-point Maximum across Vector.

FMIN (scalar): Floating-point Minimum (scalar).

FMIN (vector): Floating-point minimum (vector).

FMINNM (scalar): Floating-point Minimum Number (scalar).

FMINNM (vector): Floating-point Minimum Number (vector).

FMINNMP (scalar): Floating-point Minimum Number of Pair of elements (scalar).

FMINNMP (vector): Floating-point Minimum Number Pairwise (vector).

FMINNMV: Floating-point Minimum Number across Vector.

FMINP (scalar): Floating-point Minimum of Pair of elements (scalar).

FMINP (vector): Floating-point Minimum Pairwise (vector).

FMINV: Floating-point Minimum across Vector.

FMLA (by element): Floating-point fused Multiply-Add to accumulator (by element).

FMLA (vector): Floating-point fused Multiply-Add to accumulator (vector).

FMLS (by element): Floating-point fused Multiply-Subtract from accumulator (by element).

FMLS (vector): Floating-point fused Multiply-Subtract 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).

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).

FRSQRTE: 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).

SMOV: Signed Move vector element to general-purpose register.

SMULL, SMULL2 (by element): Signed Multiply Long (vector, by element).

SMULL, SMULL2 (vector): Signed Multiply Long (vector).

SQABS: Signed saturating Absolute value.

SQADD: Signed saturating Add.

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

SQDMLAL, SQDMLAL2 (vector): Signed saturating Doubling Multiply-Add Long.

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

SQDMLSL, SQDMLSL2 (vector): Signed saturating Doubling Multiply-Subtract Long.

SQDMULH (by element): Signed saturating Doubling Multiply returning High half (by element).

SQDMULH (vector): Signed saturating Doubling Multiply returning High half.

SQDMULL, SQDMULL2 (by element): Signed saturating Doubling Multiply Long (by element).

SQDMULL, SQDMULL2 (vector): Signed saturating Doubling Multiply Long.

SQNEG: Signed saturating Negate.

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

SQRDMLAH (vector): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

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

SQRDMLSH (vector): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).

SQRDMULH (by element): Signed saturating Rounding Doubling Multiply returning High half (by element).

SQRDMULH (vector): Signed saturating Rounding Doubling Multiply returning High half.

SQRSHL: Signed saturating Rounding Shift Left (register).

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

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

SQSHL (immediate): Signed saturating Shift Left (immediate).

SQSHL (register): Signed saturating Shift Left (register).

SQSHLU: Signed saturating Shift Left Unsigned (immediate).

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

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

SQSUB: Signed saturating Subtract.

SQXTN, SQXTN2: Signed saturating extract Narrow.

SQXTUN, SQXTUN2: Signed saturating extract Unsigned Narrow.

SRHADD: Signed Rounding Halving Add.

SRI: Shift Right and Insert (immediate).

SRSHL: Signed Rounding Shift Left (register).

SRSHR: Signed Rounding Shift Right (immediate).

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

SSHL: Signed Shift Left (register).

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

SSHR: Signed Shift Right (immediate).

SSRA: Signed Shift Right and Accumulate (immediate).

SSUBL, SSUBL2: Signed Subtract Long.

SSUBW, SSUBW2: Signed Subtract Wide.

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

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

ST2 (multiple structures): Store multiple 2-element structures from two registers.

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

ST3 (multiple structures): Store multiple 3-element structures from three registers.

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

ST4 (multiple structures): Store multiple 4-element structures from four registers.

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

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

STP (SIMD&FP): Store Pair of SIMD&FP registers.

STR (immediate, SIMD&FP): Store SIMD&FP register (immediate offset).

STR (register, SIMD&FP): Store SIMD&FP register (register offset).

STUR (SIMD&FP): Store SIMD&FP register (unscaled offset).

SUB (vector): Subtract (vector).

SUBHN, SUBHN2: Subtract returning High Narrow.

SUQADD: Signed saturating Accumulate of Unsigned value.

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

TBL: Table vector Lookup.

TBX: Table vector lookup extension.

TRN1: Transpose vectors (primary).

TRN2: Transpose vectors (secondary).

UABA: Unsigned Absolute difference and Accumulate.

UABAL, UABAL2: Unsigned Absolute difference and Accumulate Long.

UABD: Unsigned Absolute Difference (vector).

UABDL, UABDL2: Unsigned Absolute Difference Long.

UADALP: Unsigned Add and Accumulate Long Pairwise.

UADDL, UADDL2: Unsigned Add Long (vector).

UADDLP: Unsigned Add Long Pairwise.

UADDLV: Unsigned sum Long across Vector.

UADDW, UADDW2: Unsigned Add Wide.

UCVTF (scalar, fixed-point): Unsigned fixed-point Convert to Floating-point (scalar).

UCVTF (scalar, integer): Unsigned integer Convert to Floating-point (scalar).

UCVTF (vector, fixed-point): Unsigned fixed-point Convert to Floating-point (vector).

UCVTF (vector, integer): Unsigned integer Convert to Floating-point (vector).

UDOT (by element): Dot Product unsigned arithmetic (vector, by element).

UDOT (vector): Dot Product unsigned arithmetic (vector).

UHADD: Unsigned Halving Add.

UHSUB: Unsigned Halving Subtract.

UMAX: Unsigned Maximum (vector).

UMAXP: Unsigned Maximum Pairwise.

UMAXV: Unsigned Maximum across Vector.

UMIN: Unsigned Minimum (vector).

UMINP: Unsigned Minimum Pairwise.

UMINV: Unsigned Minimum across Vector.

UMLAL, UMLAL2 (by element): Unsigned Multiply-Add Long (vector, by element).

UMLAL, UMLAL2 (vector): Unsigned Multiply-Add Long (vector).

UMLSL, UMLSL2 (by element): Unsigned Multiply-Subtract Long (vector, by element).

UMLSL, UMLSL2 (vector): Unsigned Multiply-Subtract Long (vector).

UMOV: Unsigned Move vector element to general-purpose register.

UMULL, UMULL2 (by element): Unsigned Multiply Long (vector, by element).

UMULL, UMULL2 (vector): Unsigned Multiply long (vector).

UQADD: Unsigned saturating Add.

UQRSHL: Unsigned saturating Rounding Shift Left (register).

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

UQSHL (immediate): Unsigned saturating Shift Left (immediate).

UQSHL (register): Unsigned saturating Shift Left (register).

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

UQSUB: Unsigned saturating Subtract.

UQXTN, UQXTN2: Unsigned saturating extract Narrow.

URECPE: Unsigned Reciprocal Estimate.

URHADD: Unsigned Rounding Halving Add.

URSHL: Unsigned Rounding Shift Left (register).

URSHR: Unsigned Rounding Shift Right (immediate).

URSQRTE: Unsigned Reciprocal Square Root Estimate.

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

USHL: Unsigned Shift Left (register).

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

USHR: Unsigned Shift Right (immediate).

USQADD: Unsigned saturating Accumulate of Signed value.

USRA: Unsigned Shift Right and Accumulate (immediate).

USUBL, USUBL2: Unsigned Subtract Long.

USUBW, USUBW2: Unsigned Subtract Wide.

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

UZP1: Unzip vectors (primary).

UZP2: Unzip vectors (secondary).

XAR: Exclusive OR and Rotate.

XTN, XTN2: Extract Narrow.

ZIP1: Zip vectors (primary).

ZIP2: Zip vectors (secondary).

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ABS

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

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

It has encodings from 2 classes: Scalar and Vector

Scalar

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

Scalar

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

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

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

Vector

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

Vector

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

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

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

Assembler Symbols

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

size	<V>
0x	RESERVED
10	RESERVED
11	D

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

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

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

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

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

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

Operation

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

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

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADC

Add with Carry adds two register values and the Carry flag value, and writes the result to the destination register.

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

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

if sub_op then
    operand2 = NOT(operand2);

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

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

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADCS

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

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

if sub_op then
    operand2 = NOT(operand2);

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

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

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (extended register)

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Operation

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

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

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

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

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


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (immediate)

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

This instruction is used by the alias MOV (to/from SP).

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
    when '1x' ReservedValue();
```

Assembler Symbols

<Wd|WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn|WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift>	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift”:

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

Alias Conditions

Alias	Is preferred when
MOV (to/from SP)	`shift == '00' && imm12 == '000000000000' && (Rd == '11111' || Rn == '11111')`

Operation

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

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

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

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

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (shifted register)

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	0	1	0	1	1	shift		0	Rm					imm6						Rn					Rd				
	op	S																													

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

Operation

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

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

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

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

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (vector)

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

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

It has encodings from 2 classes: Scalar and Vector

Scalar

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

Scalar

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

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

Vector

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

Vector

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

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

Assembler Symbols

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

size	<V>
0x	RESERVED
10	RESERVED
11	D

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

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

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

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

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

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

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

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

Operation

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

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

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDHN, ADDHN2

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

The results are truncated. For rounded results, see *RADDHN*.

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

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

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

Three registers, not all the same type

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

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

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

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

Assembler Symbols

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

Q	2
0	[absent]
1	[present]

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

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

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

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

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

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

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

Operation

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

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

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDP (scalar)

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

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

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

Advanced SIMD

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

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

if size != '11' then ReservedValue();

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

ReduceOp op = ReduceOp_ADD;
```

Assembler Symbols

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

size	<V>
0x	RESERVED
10	RESERVED
11	D

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

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

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

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

Operation

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDP (vector)

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

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

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

Three registers of the same type

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

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

Assembler Symbols

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

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

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

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

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

Operation

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

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

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDS (extended register)

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

This instruction is used by the alias CMN (extended register).

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Alias Conditions

Alias	Is preferred when
CMN (extended register)	`Rd == '11111'`

Operation

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

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

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

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

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDS (immediate)

Add (immediate), setting flags, adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (immediate).

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12 : Zeros(12), datasize);
    when '1x' ReservedValue();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn|WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift>	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift”:

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

Alias Conditions

Alias	Is preferred when
CMN (immediate)	`Rd == '11111'`

Operation

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

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

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

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

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDS (shifted register)

Add (shifted register), setting flags, adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (shifted register).

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	1	0	1	0	1	1	shift		0	Rm	imm6	Rn	Rd
	op	S												

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

Alias Conditions

Alias	Is preferred when
CMN (shifted register)	`Rd == '11111'`

Operation

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

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

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

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

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDV

Add across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

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

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

Advanced SIMD

```
ADDV <V><d>, <Vn>.<T>
```

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

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

ReduceOp op = ReduceOp_ADD;
```

Assembler Symbols

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

size	<V>
00	B
01	H
10	S
11	RESERVED

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

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

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

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

Operation

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


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADR

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

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

Literal

```
ADR <Xd>, <label>
```

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

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

Assembler Symbols

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

Operation

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

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

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADRP

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

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

Literal

```
ADRP <Xd>, <label>
```

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

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

Assembler Symbols

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

Operation

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

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

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESD

AES single round decryption.

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

Advanced SIMD

```
AESD <Vd>.16B, <Vn>.16B
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
if decrypt then
    result = AESInvSubBytes(AESInvShiftRows(result));
else
    result = AESSubBytes(AESShiftRows(result));

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESE

AES single round encryption.

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

Advanced SIMD

```
AESE <Vd>.16B, <Vn>.16B
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
if decrypt then
    result = AESInvSubBytes(AESInvShiftRows(result));
else
    result = AESSubBytes(AESShiftRows(result));

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESIMC

AES inverse mix columns.

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

Advanced SIMD

```
AESIMC <Vd>.16B, <Vn>.16B
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

Assembler Symbols

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

Operation

```
CheckCryptoEnabled64();

bits(128) operand = V[n];
bits(128) result;
if decrypt then
    result = AESInvMixColumns(operand);
else
    result = AESMixColumns(operand);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESMC

AES mix columns.

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

Advanced SIMD

```
AESMC <Vd>.16B, <Vn>.16B
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if ! HaveCryptoExt() then UnallocatedEncoding();
boolean decrypt = (D == '1');
```

Assembler Symbols

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

Operation

```
CheckCryptoEnabled64();

bits(128) operand = V[n];
bits(128) result;
if decrypt then
    result = AESInvMixColumns(operand);
else
    result = AESMixColumns(operand);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND (vector)

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

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

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

Three registers of the same type

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

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

boolean invert = (size<0> == '1');
LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

Assembler Symbols

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

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

Q	<T>
0	8B
1	16B

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND
        result = operand1 AND operand2;
    when LogicalOp_ORR
        result = operand1 OR operand2;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

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

32-bit (sf == 0 && N == 0)

```
AND <Wd|WSP>, <Wn>, #<imm>
```

64-bit (sf == 1)

```
AND <Xd|SP>, <Xn>, #<imm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Assembler Symbols

<Wd|WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

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

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND (shifted register)

Bitwise AND (shifted register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 27 26 25 24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0 0	0 1 0 1 0	shift	0	Rm	imm6	Rn	Rd
	opc			N				

32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

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

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

Assembler Symbols

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

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

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

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

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

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

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

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

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

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

Operation

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

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ANDS (immediate)

Bitwise AND (immediate), setting flags, performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (immediate).

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

32-bit (sf == 0 && N == 0)

```
ANDS <Wd>, <Wn>, #<imm>
```

64-bit (sf == 1)

```
ANDS <Xd>, <Xn>, #<imm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
	For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Alias Conditions

Alias	Is preferred when
TST (immediate)	`Rd == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ANDS (shifted register)

Bitwise AND (shifted register), setting flags, performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (shifted register).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	0	1	0	1	0	shift		0	Rm					imm6						Rn					Rd				
	opc									N																					

32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

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

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

Assembler Symbols

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

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

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

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

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

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

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

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

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

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

Alias Conditions

Alias	Is preferred when
TST (shifted register)	`Rd == '11111'`

Operation

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

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of ASRV. This means:

- The encodings in this description are named to match the encodings of ASRV.
- The description of ASRV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0	1	0	Rn					Rd				
																				op2											

32-bit (sf == 0)

```
ASR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
ASRV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
ASR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
ASRV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of ASRV gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of the sign bit in the upper bits and zeros in the lower bits, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

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

32-bit (sf == 0 && N == 0 && imms == 011111)

```
ASR <Wd>, <Wn>, #<shift>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1 && imms == 111111)

```
ASR <Xd>, <Xn>, #<shift>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift>	For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASRV

Arithmetic Shift Right Variable shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ASR (register).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0	1	0	Rn					Rd				
																				op2											

32-bit (sf == 0)

```
ASRV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
ASRV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AT

Address Translate. For more information, see *A64 system instructions for address translation*.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	1	op1			0	1	1	1	1	0	0	x	op2			Rt				
										L							CRn				CRm									

System

```
AT <at_op>, <Xt>
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1,'0111',CRm,op2) == Sys_AT`.

Assembler Symbols

<at_op> Is an AT instruction name, as listed for the AT system instruction group, encoded in “op1:CRm<0>:op2”:

op1	CRm<0>	op2	<at_op>	Architectural Feature
000	0	000	S1E1R	-
000	0	001	S1E1W	-
000	0	010	S1E0R	-
000	0	011	S1E0W	-
000	1	000	S1E1RP	ARMv8.2-ATS1E1
000	1	001	S1E1WP	ARMv8.2-ATS1E1
100	0	000	S1E2R	-
100	0	001	S1E2W	-
100	0	100	S12E1R	-
100	0	101	S12E1W	-
100	0	110	S12E0R	-
100	0	111	S12E0W	-
110	0	000	S1E3R	-
110	0	001	S1E3W	-

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

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

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AUTDA, AUTDZA

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for `AUTDA`.
- The value zero, for `AUTDZA`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Integer

(ARMv8.3)

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

AUTDA (Z == 0)

```
AUTDA <Xd>, <Xn|SP>
```

AUTDZA (Z == 1 && Rn == 11111)

```
AUTDZA <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UnallocatedEncoding();

if Z == '0' then // AUTDA
    if n == 31 then source_is_sp = TRUE;
else // AUTDZA
    if n != 31 then UnallocatedEncoding();
```

Assembler Symbols

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

Operation

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AUTDB, AUTDZB

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for `AUTDB`.
- The value zero, for `AUTDZB`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Integer

(ARMv8.3)

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

AUTDB (Z == 0)

```
AUTDB <Xd>, <Xn|SP>
```

AUTDZB (Z == 1 && Rn == 11111)

```
AUTDZB <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UnallocatedEncoding();

if Z == '0' then // AUTDB
    if n == 31 then source_is_sp = TRUE;
else // AUTDZB
    if n != 31 then UnallocatedEncoding();
```

Assembler Symbols

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

Operation

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

Authenticate Instruction address, using key A. This instruction authenticates an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by <Xd> for `AUTIA` and `AUTIZA`.
- In X17, for `AUTIA1716`.
- In X30, for `AUTIASP` and `AUTIAZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for `AUTIA`.
- The value zero, for `AUTIZA` and `AUTIAZ`.
- In X16, for `AUTIA1716`.
- In SP, for `AUTIASP`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

It has encodings from 2 classes: Integer and System

Integer

(ARMv8.3)

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

AUTIA (Z == 0)

```
AUTIA <Xd>, <Xn|SP>
```

AUTIZA (Z == 1 && Rn == 11111)

```
AUTIZA <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UnallocatedEncoding();

if Z == '0' then // AUTIA
    if n == 31 then source_is_sp = TRUE;
else // AUTIZA
    if n != 31 then UnallocatedEncoding();
```

System

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	1	0	x	1	1	1	1	1
																					CRm				op2						

AUTIA1716 (CRm == 0001 && op2 == 100)

```
AUTIA1716
```

AUTIASP (CRm == 0011 && op2 == 101)

```
AUTIASP
```

AUTIAZ (CRm == 0011 && op2 == 100)

```
AUTIAZ
```

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 100' // AUTIAZ
        d = 30;
        n = 31;
    when '0011 101' // AUTIASP
        d = 30;
        source_is_sp = TRUE;
    when '0001 100' // AUTIA1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 010' SEE "PACIB";
    when '0001 110' SEE "AUTIB";
    when '0011 00x' SEE "PACIA";
    when '0011 01x' SEE "PACIB";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";
```

Assembler Symbols

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

Operation

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


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

Authenticate Instruction address, using key B. This instruction authenticates an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by <Xd> for `AUTIB` and `AUTIZB`.
- In X17, for `AUTIB1716`.
- In X30, for `AUTIBSP` and `AUTIBZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for `AUTIB`.
- The value zero, for `AUTIZB` and `AUTIBZ`.
- In X16, for `AUTIB1716`.
- In SP, for `AUTIBSP`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

It has encodings from 2 classes: Integer and System

Integer

(ARMv8.3)

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

AUTIB (Z == 0)

```
AUTIB <Xd>, <Xn|SP>
```

AUTIZB (Z == 1 && Rn == 11111)

```
AUTIZB <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UnallocatedEncoding();

if Z == '0' then // AUTIB
    if n == 31 then source_is_sp = TRUE;
else // AUTIZB
    if n != 31 then UnallocatedEncoding();
```

System

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	1	1	x	1	1	1	1	1
																					CRm				op2						

AUTIB1716 (CRm == 0001 && op2 == 110)

```
AUTIB1716
```

AUTIBSP (CRm == 0011 && op2 == 111)

```
AUTIBSP
```

AUTIBZ (CRm == 0011 && op2 == 110)

```
AUTIBZ
```

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 110' // AUTIBZ
        d = 30;
        n = 31;
    when '0011 111' // AUTIBSP
        d = 30;
        source_is_sp = TRUE;
    when '0001 110' // AUTIB1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 010' SEE "PACIB";
    when '0001 100' SEE "AUTIA";
    when '0011 00x' SEE "PACIA";
    when '0011 01x' SEE "PACIB";
    when '0011 10x' SEE "AUTIA";
    when '0000 111' SEE "XPACLRI";
```

Assembler Symbols

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

Operation

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


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

B.cond

Branch conditionally to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	imm19																			0	cond			

19-bit signed PC-relative branch offset

```
B.<cond> <label>
```

```
bits(64) offset = SignExtend(imm19:'00', 64);
bits(4) condition = cond;
```

Assembler Symbols

<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.
<label>	Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
if ConditionHolds(condition) then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

B

Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

31	30	29	28	27	26	25	24	23	22	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	0	1	imm26																									
op																															

26-bit signed PC-relative branch offset

```
B <label>
```

```
BranchType branch_type = if op == '1' then BranchType_CALL else BranchType_JMP;
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler Symbols

<label>	Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

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

BranchTo(PC[] + offset, branch_type);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BCAX

Bit Clear and Exclusive OR performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD

(ARMv8.2)

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

Advanced SIMD

```
BCAX <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B
```

```
if !HaveCryptoExt2() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer a = UInt(Ra);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va>	Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Va = V[a];
V[d] = Vn EOR (Vm AND NOT(Va));
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFC

Bitfield Clear, leaving other bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

Leaving other bits unchanged

(ARMv8.2)

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

32-bit (sf == 0 && N == 0)

```
BFC <Wd>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Wd>, WZR, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit (sf == 1 && N == 1)

```
BFC <Xd>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Xd>, XZR, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

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.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFI

Bitfield Insert copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, leaving other bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	1	1	0	0	1	1	0	N	immr	imms	!= 11111	Rd
	opc											Rn	

32-bit (sf == 0 && N == 0)

```
BFI <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit (sf == 1 && N == 1)

```
BFI <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFM

Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, leaving other bits unchanged.

This instruction is used by the aliases BFC, BFI, and BFXIL.

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

32-bit (sf == 0 && N == 0)

```
BFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit (sf == 1 && N == 1)

```
BFM <Xd>, <Xn>, #<immr>, #<imms>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

boolean inzero;
boolean extend;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

case opc of
    when '00' inzero = TRUE;  extend = TRUE;    // SBFM
    when '01' inzero = FALSE; extend = FALSE;   // BFM
    when '10' inzero = TRUE;  extend = FALSE;   // UBFM
    when '11' UnallocatedEncoding();

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

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

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
<imms>	For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Is preferred when
BFC	`Rn == '11111' && UInt(imms) < UInt(immr)`
BFI	`Rn != '11111' && UInt(imms) < UInt(immr)`
BFXIL	`UInt(imms) >= UInt(immr)`

Operation

```
bits(datasize) dst = if inzero then Zeros() else X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// determine extension bits (sign, zero or dest register)
bits(datasize) top = if extend then Replicate(src<S>) else dst;

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFXIL

Bitfield extract and insert at low end copies any number of low-order bits from a source register into the same number of adjacent bits at the low end in the destination register, leaving other bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

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

32-bit (sf == 0 && N == 0)

```
BFXIL <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

64-bit (sf == 1 && N == 1)

```
BFXIL <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC (vector, immediate)

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd				
		op														cmode															

16-bit (cmode == 10x1)

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit (cmode == 0xx1)

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

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

<T> For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC (vector, register)

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

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

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

Three registers of the same type

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

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

boolean invert = (size<0> == '1');
LogicalOp op = if size<1> == '1' then LogicalOp_ORR else LogicalOp_AND;
```

Assembler Symbols

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

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

Q	<T>
0	8B
1	16B

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND
        result = operand1 AND operand2;
    when LogicalOp_ORR
        result = operand1 OR operand2;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC (shifted register)

Bitwise Bit Clear (shifted register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 27 26 25 24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0 0	0 1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc			N				

32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

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

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

Assembler Symbols

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

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

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

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

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

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

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

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

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

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

Operation

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

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags, performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	0	1	0	1	0	shift		1	Rm					imm6						Rn					Rd				
	opc									N																					

32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

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

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
boolean invert = (N == '1');
```

Assembler Symbols

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

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

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

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

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

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

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

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

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

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

Operation

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

if invert then operand2 = NOT(operand2);

case op of
    when LogicalOp_AND result = operand1 AND operand2;
    when LogicalOp_ORR result = operand1 OR  operand2;
    when LogicalOp_EOR result = operand1 EOR operand2;

if setflags then
    PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIF

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	Rm					0	0	0	1	1	1	Rn					Rd				
								opc2																							

Three registers of the same type

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

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

VBitOp op;

case opc2 of
    when '00' op = VBitOp_VEOR;
    when '01' op = VBitOp_VBSL;
    when '10' op = VBitOp_VBIT;
    when '11' op = VBitOp_VBIF;
```

Assembler Symbols

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

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

Q	<T>
0	8B
1	16B

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

case op of
    when VBitOp_VEOR
        operand1 = V[m];
        operand2 = Zeros();
        operand3 = Ones();
    when VBitOp_VBSL
        operand1 = V[m];
        operand2 = operand1;
        operand3 = V[d];
    when VBitOp_VBIT
        operand1 = V[d];
        operand2 = operand1;
        operand3 = V[m];
    when VBitOp_VBIF
        operand1 = V[d];
        operand2 = operand1;
        operand3 = NOT(V[m]);

V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIT

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	0	1	Rm					0	0	0	1	1	1	Rn					Rd				
								opc2																							

Three registers of the same type

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

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

VBitOp op;

case opc2 of
    when '00' op = VBitOp_VEOR;
    when '01' op = VBitOp_VBSL;
    when '10' op = VBitOp_VBIT;
    when '11' op = VBitOp_VBIF;
```

Assembler Symbols

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

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

Q	<T>
0	8B
1	16B

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

case op of
    when VBitOp_VEOR
        operand1 = V[m];
        operand2 = Zeros();
        operand3 = Ones();
    when VBitOp_VBSL
        operand1 = V[m];
        operand2 = operand1;
        operand3 = V[d];
    when VBitOp_VBIT
        operand1 = V[d];
        operand2 = operand1;
        operand3 = V[m];
    when VBitOp_VBIF
        operand1 = V[d];
        operand2 = operand1;
        operand3 = NOT(V[m]);

V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BL

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	imm26																									
op																															

26-bit signed PC-relative branch offset

```
BL <label>
```

```
BranchType branch_type = if op == '1' then BranchType_CALL else BranchType_JMP;
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler Symbols

<label>	Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

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

BranchTo(PC[] + offset, branch_type);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

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

Integer

```
BLR <Xn>
```

```
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UnallocatedEncoding();
elsif pac && !HavePACExt() then
    UnallocatedEncoding();

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n !=31 then UnallocatedEncoding();
        n = 30;
        source_is_sp = TRUE;
```

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

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

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

if branch_type == BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4. The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP> for `BLRAA` and `BLRAB`.
- The value zero, for `BLRAAZ` and `BLRABZ`.

Key A is used for `BLRAA` and `BLRAAZ`, and key B is used for `BLRAB` and `BLRABZ`.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated. The authenticated address is not written back to the general-purpose register.

Integer

(ARMv8.3)

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

Key A, zero modifier (Z == 0 && M == 0 && Rm == 11111)

```
BLRAAZ <Xn>
```

Key A, register modifier (Z == 1 && M == 0)

```
BLRAA <Xn>, <Xm|SP>
```

Key B, zero modifier (Z == 0 && M == 1 && Rm == 11111)

```
BLRABZ <Xn>
```

Key B, register modifier (Z == 1 && M == 1)

```
BLRAB <Xn>, <Xm|SP>
```

```
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UnallocatedEncoding();
elsif pac && !HavePACExt() then
    UnallocatedEncoding();

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n !=31 then UnallocatedEncoding();
        n = 30;
        source_is_sp = TRUE;
```

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.
<Xm|SP>	Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

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

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

if branch_type == BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BR

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

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

Integer

```
BR <Xn>
```

```
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UnallocatedEncoding();
elsif pac && !HavePACExt() then
    UnallocatedEncoding();

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n !=31 then UnallocatedEncoding();
        n = 30;
        source_is_sp = TRUE;
```

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

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

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

if branch_type == BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP> for `BRAA` and `BRAB`.
- The value zero, for `BRAAZ` and `BRABZ`.

Key A is used for `BRAA` and `BRAAZ`, and key B is used for `BRAB` and `BRABZ`.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

Integer

(ARMv8.3)

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

Key A, zero modifier (Z == 0 && M == 0 && Rm == 11111)

```
BRAAZ <Xn>
```

Key A, register modifier (Z == 1 && M == 0)

```
BRAA <Xn>, <Xm|SP>
```

Key B, zero modifier (Z == 0 && M == 1 && Rm == 11111)

```
BRABZ <Xn>
```

Key B, register modifier (Z == 1 && M == 1)

```
BRAB <Xn>, <Xm|SP>
```

```
integer n = UInt(Rn);
BranchType branch_type;
integer m = UInt(Rm);
boolean pac = (A == '1');
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !pac && m != 0 then
    UnallocatedEncoding();
elsif pac && !HavePACExt() then
    UnallocatedEncoding();

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();

if pac then
    if Z == '0' && m != 31 then
        UnallocatedEncoding();

    if branch_type == BranchType_RET then
        if n !=31 then UnallocatedEncoding();
        n = 30;
        source_is_sp = TRUE;
```

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.
<Xm|SP>	Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

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

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

if branch_type == BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BRK

Breakpoint instruction generates a Breakpoint Instruction exception. The PE records the exception in *ESR_ELx*, using the EC value 0x3c, and captures the value of the immediate argument in *ESR_ELx*.ISS.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	1	imm16																0	0	0	0	0

System

```
BRK #<imm>
```

```
bits(16) comment = imm16;
```

Assembler Symbols

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

Operation

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BSL

Bitwise Select. This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	Rm					0	0	0	1	1	1	Rn					Rd				
								opc2																							

Three registers of the same type

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

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

VBitOp op;

case opc2 of
    when '00' op = VBitOp_VEOR;
    when '01' op = VBitOp_VBSL;
    when '10' op = VBitOp_VBIT;
    when '11' op = VBitOp_VBIF;
```

Assembler Symbols

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

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

Q	<T>
0	8B
1	16B

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

case op of
    when VBitOp_VEOR
        operand1 = V[m];
        operand2 = Zeros();
        operand3 = Ones();
    when VBitOp_VBSL
        operand1 = V[m];
        operand2 = operand1;
        operand3 = V[d];
    when VBitOp_VBIT
        operand1 = V[d];
        operand2 = operand1;
        operand3 = V[m];
    when VBitOp_VBIF
        operand1 = V[d];
        operand2 = operand1;
        operand3 = NOT(V[m]);

V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- `CASA` and `CASAL` load from memory with acquire semantics.
- `CASL` and `CASAL` store to memory with release semantics.
- `CAS` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, or <Xs>, is restored to the value held in the register before the instruction was executed.

No offset

(ARMv8.1)

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

32-bit, acquire (size == 10 && L == 1 && o0 == 0)

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

32-bit, acquire and release (size == 10 && L == 1 && o0 == 1)

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

32-bit, no memory ordering (size == 10 && L == 0 && o0 == 0)

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

32-bit, release (size == 10 && L == 0 && o0 == 1)

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

64-bit, acquire (size == 11 && L == 1 && o0 == 0)

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

64-bit, acquire and release (size == 11 && L == 1 && o0 == 1)

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

64-bit, no memory ordering (size == 11 && L == 0 && o0 == 0)

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

64-bit, release (size == 11 && L == 0 && o0 == 1)

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

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, datasize DIV 8, stacctype] = newvalue;

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset

(ARMv8.1)

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

Acquire (L == 1 && o0 == 0)

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

Acquire and release (L == 1 && o0 == 1)

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

No memory ordering (L == 0 && o0 == 0)

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

Release (L == 0 && o0 == 1)

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

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, datasize DIV 8, stacctype] = newvalue;

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CAS has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset

(ARMv8.1)

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

Acquire (L == 1 && o0 == 0)

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

Acquire and release (L == 1 && o0 == 1)

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

No memory ordering (L == 0 && o0 == 0)

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

Release (L == 0 && o0 == 1)

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

```
if !HaveAtomicExt() then UnallocatedEncoding();

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, datasize DIV 8, stacctype] = newvalue;

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- `CASPA` and `CASPAL` load from memory with acquire semantics.
- `CASPL` and `CASPAL` store to memory with release semantics.
- `CAS` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is <Ws> and <W(s+1)>, or <Xs> and <X(s+1)>, are restored to the values held in the registers before the instruction was executed.

No offset

(ARMv8.1)

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

32-bit, acquire (sz == 0 && L == 1 && o0 == 0)

```
CASPA <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit, acquire and release (sz == 0 && L == 1 && o0 == 1)

```
CASPAL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit, no memory ordering (sz == 0 && L == 0 && o0 == 0)

```
CASP <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit, release (sz == 0 && L == 0 && o0 == 1)

```
CASPL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

64-bit, acquire (sz == 1 && L == 1 && o0 == 0)

```
CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit, acquire and release (sz == 1 && L == 1 && o0 == 1)

```
CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit, no memory ordering (sz == 1 && L == 0 && o0 == 0)

```
CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit, release (sz == 1 && L == 0 && o0 == 1)

```
CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
if Rs<0> == '1' then UnallocatedEncoding();
if Rt<0> == '1' then UnallocatedEncoding();

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

integer datasize = 32 << UInt(sz);
integer regsize = datasize;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field.
<W(s+1)>	Is the 32-bit name of the second general-purpose register to be compared and loaded.
<Wt>	Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field.
<W(t+1)>	Is the 32-bit name of the second general-purpose register to be conditionally stored.
<Xs>	Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field.
<X(s+1)>	Is the 64-bit name of the second general-purpose register to be compared and loaded.
<Xt>	Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field.
<X(t+1)>	Is the 64-bit name of the second general-purpose register to be conditionally stored.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

bits(datasize) s1 = X[s];
bits(datasize) s2 = X[s+1];
bits(datasize) t1 = X[t];
bits(datasize) t2 = X[t+1];
comparevalue = if BigEndian() then s1:s2 else s2:s1;
newvalue     = if BigEndian() then t1:t2 else t2:t1;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data =  Mem[address, (2 * datasize) DIV 8, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, (2 * datasize) DIV 8, stacctype] = newvalue;

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	0	1	0	1	imm19																			Rt				
							op																								

32-bit (sf == 0)

```
CBNZ <Wt>, <label>
```

64-bit (sf == 1)

```
CBNZ <Xt>, <label>
```

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<label>	Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(datasize) operand1 = X[t];

if IsZero(operand1) == iszero then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	0	1	0	0	imm19																			Rt				
							op																								

32-bit (sf == 0)

```
CBZ <Wt>, <label>
```

64-bit (sf == 1)

```
CBZ <Xt>, <label>
```

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
boolean iszero = (op == '0');
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<label>	Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(datasize) operand1 = X[t];

if IsZero(operand1) == iszero then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CCMN (immediate)

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

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12	11	10	9 8 7 6 5	4	3 2 1 0
sf	0	1	1	1	0	1	0	0	1	0	imm5	cond	1	0	Rn	0	nzcv
	op																

32-bit (sf == 0)

```
CCMN <Wn>, #<imm>, #<nzcv>, <cond>
```

64-bit (sf == 1)

```
CCMN <Xn>, #<imm>, #<nzcv>, <cond>
```

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

Assembler Symbols

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

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;
bit carry_in = '0';

if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CCMN (register)

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bit carry_in = '0';

if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CCMP (immediate)

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

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12	11	10	9 8 7 6 5	4	3 2 1 0
sf	1	1	1	1	0	1	0	0	1	0	imm5	cond	1	0	Rn	0	nzcv
	op																

32-bit (sf == 0)

```
CCMP <Wn>, #<imm>, #<nzcv>, <cond>
```

64-bit (sf == 1)

```
CCMP <Xn>, #<imm>, #<nzcv>, <cond>
```

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

Assembler Symbols

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

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;
bit carry_in = '0';

if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CCMP (register)

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bit carry_in = '0';

if ConditionHolds(condition) then
    if sub_op then
        operand2 = NOT(operand2);
        carry_in = '1';
    (-, flags) = AddWithCarry(operand1, operand2, carry_in);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CINC

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

This is an alias of CSINC. This means:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	1	0	1	0	1	0	0	!= 11111	!= 111x	0	1	!= 11111	Rd
op											Rm	cond		o2	Rn	

32-bit (sf == 0)

```
CINC <Wd>, <Wn>, <cond>
```

is equivalent to

```
CSINC <Wd>, <Wn>, <Wn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

64-bit (sf == 1)

```
CINC <Xd>, <Xn>, <cond>
```

is equivalent to

```
CSINC <Xd>, <Xn>, <Xn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<cond>	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CINV

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

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
sf	1	0	1	1	0	1	0	1	0	0	!= 11111	!= 111x	0	0	!= 11111	Rd
op											Rm	cond		o2	Rn	

32-bit (sf == 0)

```
CINV <Wd>, <Wn>, <cond>
```

is equivalent to

```
CSINV <Wd>, <Wn>, <Wn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

64-bit (sf == 1)

```
CINV <Xd>, <Xn>, <cond>
```

is equivalent to

```
CSINV <Xd>, <Xn>, <Xn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<cond>	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLREX

Clear Exclusive clears the local monitor of the executing PE.

31	30	29	28	27	26	25	24	23	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				0	1	0	1	1	1	1	1

System

```
CLREX {#<imm>}
```

```
// CRm field is ignored
```

Assembler Symbols

<imm>	Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

```
ClearExclusiveLocal(ProcessorID());
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLS (vector)

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The count does not include the most significant bit itself.

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

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

Vector

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

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

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

Assembler Symbols

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

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

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

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

Operation

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

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLS

Count leading sign bits: `Rd = CLS(Rn)`.

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

32-bit (sf == 0)

```
CLS <Wd>, <Wn>
```

64-bit (sf == 1)

```
CLS <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
CountOp opcode = if op == '0' then CountOp_CLZ else CountOp_CLS;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
integer result;
bits(datasize) operand1 = X[n];

if opcode == CountOp_CLZ then
    result = CountLeadingZeroBits(operand1);
else
    result = CountLeadingSignBits(operand1);

X[d] = result<datasize-1:0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLZ (vector)

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

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

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

Vector

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

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

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

Assembler Symbols

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

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

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

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

Operation

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

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLZ

Count leading zero bits: `Rd = CLZ(Rn)`.

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

32-bit (sf == 0)

```
CLZ <Wd>, <Wn>
```

64-bit (sf == 1)

```
CLZ <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
CountOp opcode = if op == '0' then CountOp_CLZ else CountOp_CLS;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
integer result;
bits(datasize) operand1 = X[n];

if opcode == CountOp_CLZ then
    result = CountLeadingZeroBits(operand1);
else
    result = CountLeadingSignBits(operand1);

X[d] = result<datasize-1:0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMEQ (register)

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

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

It has encodings from 2 classes: Scalar and Vector

Scalar

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

Scalar

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

Vector

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

Vector

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

Assembler Symbols

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

size	<V>
0x	RESERVED
10	RESERVED
11	D

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

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

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

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

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

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

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

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

Operation

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

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMEQ (zero)

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

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

It has encodings from 2 classes: Scalar and Vector

Scalar

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

Scalar

```
CMEQ <V><d>, <V><n>, #0
```

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

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector

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

Vector

```
CMEQ <Vd>.<T>, <Vn>.<T>, #0
```

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

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

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

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

size	<V>
0x	RESERVED
10	RESERVED
11	D

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

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

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

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

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

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

Operation

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

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMGE (register)

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

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

It has encodings from 2 classes: Scalar and Vector

Scalar

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

Scalar

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

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

Vector

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

Vector

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

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

Assembler Symbols

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

size	<V>
0x	RESERVED
10	RESERVED
11	D

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

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

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

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

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

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

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMGE (zero)

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

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

It has encodings from 2 classes: Scalar and Vector

Scalar

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

Scalar

```
CMGE <V><d>, <V><n>, #0
```

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

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector

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

Vector

```
CMGE <Vd>.<T>, <Vn>.<T>, #0
```

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

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

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

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

size	<V>
0x	RESERVED
10	RESERVED
11	D

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

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

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

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

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

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

Operation

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

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMGT (register)

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

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

It has encodings from 2 classes: Scalar and Vector

Scalar

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

Scalar

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

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

Vector

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

Vector

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

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

Assembler Symbols

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

size	<V>
0x	RESERVED
10	RESERVED
11	D

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

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

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

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

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

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

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMGT (zero)

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

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

It has encodings from 2 classes: Scalar and Vector

Scalar

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

Scalar

```
CMGT <V><d>, <V><n>, #0
```

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

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector

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

Vector

```
CMGT <Vd>.<T>, <Vn>.<T>, #0
```

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

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

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

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

size	<V>
0x	RESERVED
10	RESERVED
11	D

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

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

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

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

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

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

Operation

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

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DCPS2

Debug Change PE State to EL2, when executed in Debug state:

- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

- *ELR_ELx* becomes UNKNOWN.
- *SPSR_ELx* becomes UNKNOWN.
- *ESR_ELx* becomes UNKNOWN.
- *DLR_EL0* and *DSPSR_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR_ELx*.EE.

This instruction is UNDEFINED at the following exception levels:

- All exception levels if EL2 is not implemented.
- At EL0 and EL1 in Secure state if EL2 is implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16																0	0	0	1	0
																														LL	

System

```
DCPS2 {#<imm>}
```

```
if !Halted() then AArch64.UndefinedFault();
```

Assembler Symbols

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

Operation

```
DCPSInstruction(LL);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier*.

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

System

```
DSB <option>|#<imm>
```

```
MBReqDomain domain;
MBReqTypes types;

case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
    otherwise
        types = MBReqTypes_All;
        domain = MBReqDomain_FullSystem;
```

Assembler Symbols

<option> Specifies the limitation on the barrier operation. Values are:

SY
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST
Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD
Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type before the barrier instruction, and reads and writes are the required type after the barrier instruction. Encoded as CRm = 0b0110.

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores PSTATE from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See Illegal return events from AArch64 state.

ERET is UNDEFINED at EL0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
																				A	M			Rn					op4		

System

```
ERET
```

```
if PSTATE.EL == EL0 then UnallocatedEncoding();
```

Operation

```
AArch64.CheckForERetTrap(FALSE, TRUE);
bits(64) target = ELR[];

AArch64.ExceptionReturn(target, SPSR[]);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DCPS2

Debug Change PE State to EL2, when executed in Debug state:

- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

- *ELR_ELx* becomes UNKNOWN.
- *SPSR_ELx* becomes UNKNOWN.
- *ESR_ELx* becomes UNKNOWN.
- *DLR_EL0* and *DSPSR_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR_ELx*.EE.

This instruction is UNDEFINED at the following exception levels:

- All exception levels if EL2 is not implemented.
- At EL0 and EL1 in Secure state if EL2 is implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16																0	0	0	1	0
																														LL	

System

```
DCPS2 {#<imm>}
```

```
if !Halted() then AArch64.UndefinedFault();
```

Assembler Symbols

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

Operation

```
DCPSInstruction(LL);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DCPS3

Debug Change PE State to EL3, when executed in Debug state:

- If executed at EL3 selects SP_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

- *ELR_EL3* becomes UNKNOWN.
- *SPSR_EL3* becomes UNKNOWN.
- *ESR_EL3* becomes UNKNOWN.
- *DLR_EL0* and *DSPSR_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR_EL3*.EE.

This instruction is UNDEFINED at all exception levels if either:

- *EDSCR.SDD* == 1.
- EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16																0	0	0	1	1
																														LL	

System

```
DCPS3 {#<imm>}
```

```
if !Halted() then AArch64.UndefinedFault();
```

Assembler Symbols

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

Operation

```
DCPSInstruction(LL);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier*.

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

System

```
DMB <option>|#<imm>
```

```
MBReqDomain domain;
MBReqTypes types;

case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
    otherwise
        types = MBReqTypes_All;
        domain = MBReqDomain_FullSystem;
```

Assembler Symbols

<option> Specifies the limitation on the barrier operation. Values are:

SY
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST
Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD
Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type before the barrier instruction, and reads and writes are the required type after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD
Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* or see *Data Synchronization Barrier (DSB)*.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
DataMemoryBarrier(domain, types);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DRPS

Debug restore process state.

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

System

```
DRPS
```

```
if !Halted() || PSTATE.EL == EL0 then UnallocatedEncoding();
```

Operation

```
DRPSInstruction();
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier*.

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

System

```
DSB <option>|#<imm>
```

```
MBReqDomain domain;
MBReqTypes types;

case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

case CRm<1:0> of
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
    otherwise
        types = MBReqTypes_All;
        domain = MBReqDomain_FullSystem;
```

Assembler Symbols

<option> Specifies the limitation on the barrier operation. Values are:

SY
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST
Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD
Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type before the barrier instruction, and reads and writes are the required type after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD
Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* or see *Data Synchronization Barrier (DSB)*.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
DataSynchronizationBarrier(domain, types);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DUP (element)

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

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

This instruction is used by the alias MOV (scalar).

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	imm5					0	0	0	0	0	1	Rn					Rd				

Scalar

```
DUP <V><d>, <Vn>.<T>[<index>]
```

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

integer size = LowestSetBit(imm5);
if size > 3 then UnallocatedEncoding();

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	imm5					0	0	0	0	0	1	Rn					Rd				

Vector

```
DUP <Vd>.<T>, <Vn>.<Ts>[<index>]
```

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

integer size = LowestSetBit(imm5);
if size > 3 then UnallocatedEncoding();

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then ReservedValue();
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<T>	For the scalar variant: is the element width specifier, encoded in “imm5”:

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

For the vector variant: is an arrangement specifier, encoded in “imm5:Q”:

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<V> Is the destination width specifier, encoded in “imm5”:

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

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

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(datasize) result;
bits(esize) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DUP (general)

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register.

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

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	0	0	0	imm5	0	0	0	0	1	1	Rn	Rd

Advanced SIMD

```
DUP <Vd>.<T>, <R><n>
```

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

integer size = LowestSetBit(imm5);
if size > 3 then UnallocatedEncoding();

// imm5<4:size+1> is IGNORED

if size == 3 && Q == '0' then ReservedValue();
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

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

<T> Is an arrangement specifier, encoded in “imm5:Q”:

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

Unspecified bits in "imm5" are ignored but should be set to zero by an assembler.

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 27 26 25 24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1 0	0 1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc			N				

32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then ReservedValue();

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

Assembler Symbols

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

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

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

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

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

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

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

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

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

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

Operation

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

operand2 = NOT(operand2);

result = operand1 EOR operand2;

X[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR3

Three-way Exclusive OR performs a three-way exclusive OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD

(ARMv8.2)

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

Advanced SIMD

```
EOR3 <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B
```

```
if !HaveCryptoExt2() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer a = UInt(Ra);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va>	Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Va = V[a];
V[d] = Vn EOR Vm EOR Va;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR (vector)

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register.

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

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	0	0	0	1	Rm	0	0	0	1	1	1	Rn	Rd
								opc2											

Three registers of the same type

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

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

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

Q	<T>
0	8B
1	16B

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

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 = V[m];
operand2 = Zeros();
operand3 = Ones();
V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

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

32-bit (sf == 0 && N == 0)

```
EOR <Wd|WSP>, <Wn>, #<imm>
```

64-bit (sf == 1)

```
EOR <Xd|SP>, <Xn>, #<imm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Assembler Symbols

<Wd|WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
	For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];

result = operand1 EOR imm;

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR (shifted register)

Bitwise Exclusive OR (shifted register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30 29	28 27 26 25 24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1 0	0 1 0 1 0	shift	0	Rm	imm6	Rn	Rd
	opc			N				

32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then ReservedValue();

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

Assembler Symbols

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

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

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

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

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

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

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

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

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

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

Operation

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

result = operand1 EOR operand2;

X[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores *PSTATE* from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERET is UNDEFINED at EL0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
																				A	M			Rn					op4		

System

```
ERET
```

```
if PSTATE.EL == EL0 then UnallocatedEncoding();
```

Operation

```
AArch64.CheckForERetTrap(FALSE, TRUE);
bits(64) target = ELR[];

AArch64.ExceptionReturn(target, SPSR[]);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ERETAA, ERETAB

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores *PSTATE* from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for `ERETAA`, and key B is used for `ERETAB`.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

`ERET` is UNDEFINED at EL0.

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1
																				A				Rn					op4		

ERETAA (M == 0)

```
ERETAA
```

ERETAB (M == 1)

```
ERETAB
```

```
if PSTATE.EL == EL0 then UnallocatedEncoding();
boolean use_key_a = (M == '0');

if !HavePACExt() then
    UnallocatedEncoding();
```

Operation

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

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

AArch64.ExceptionReturn(target, SPSR[]);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR_EL1 and VDISR_EL2.

This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the ARM(R) Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for ARMv8-A architecture profile.

If the RAS Extension is not implemented, this instruction executes as a NOP.

System

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	0	1	1	1	1	1
																					CRm				op2						

System

```
ESB
```

```
if !HaveRASExt() then EndOfInstruction();
```

Operation

```
ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);
AArch64.ESBOperation();
if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
TakeUnmaskedSErrorInterrupts();
```

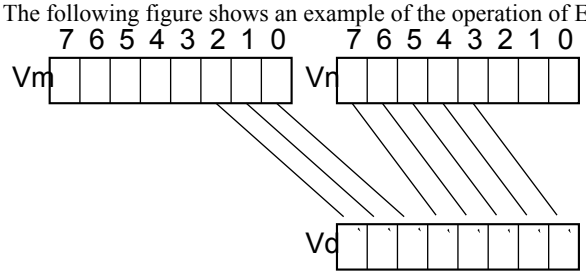
Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EXT

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows an example of the operation of EXT doubleword operation for Q = 0 and imm4<2:0> = 3.



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

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12 11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	0	0	0	0	Rm	0	imm4	0	Rn	Rd

Advanced SIMD

```
EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>
```

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

if Q == '0' && imm4<3> == '1' then UnallocatedEncoding();

integer datasize = if Q == '1' then 128 else 64;
integer position = UInt(imm4) << 3;
```

Assembler Symbols

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

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

Q	<T>
0	8B
1	16B

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

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

<index> Is the lowest numbered byte element to be extracted, encoded in “Q:imm4”:

Q	imm4<3>	<index>
0	0	imm4<2:0>
0	1	RESERVED
1	x	imm4

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m];
bits(datasize) lo = V[n];
bits(datasize*2) concat = hi:lo;

V[d] = concat<position+datasize-1:position>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EXTR

Extract register extracts a register from a pair of registers.

This instruction is used by the alias ROR (immediate).

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

32-bit (sf == 0 && N == 0 && imms == 0xxxxx)

```
EXTR <Wd>, <Wn>, <Wm>, #<lsb>
```

64-bit (sf == 1 && N == 1)

```
EXTR <Xd>, <Xn>, <Xm>, #<lsb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
integer lsb;

if N != sf then UnallocatedEncoding();
if sf == '0' && imms<5> == '1' then ReservedValue();
lsb = UInt(imms);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<lsb>	For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Is preferred when
ROR (immediate)	`Rn == Rm`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(2*datasize) concat = operand1:operand2;

result = concat<lsb+datasize-1:lsb>;

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FABD

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

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

Scalar half precision

```
FABD <Hd>, <Hn>, <Hm>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

Scalar single-precision and double-precision

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

Scalar single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

Vector half precision

(ARMv8.2)

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

Vector half precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Vector single-precision and double-precision

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

Vector single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Assembler Symbols

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

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

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

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

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

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

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

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

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

Operation

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

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FABS (vector)

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

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

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

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

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

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

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Single-precision and double-precision

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

Single-precision and double-precision

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

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

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

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

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

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

Operation

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

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FABS (scalar)

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

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

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

Half-precision (type == 11)

(ARMv8.2)

```
FABS <Hd>, <Hn>
```

Single-precision (type == 00)

```
FABS <Sd>, <Sn>
```

Double-precision (type == 01)

```
FABS <Dd>, <Dn>
```

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

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

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

Operation

```
CheckFPAdvSIMDEnabled64();

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

result = FPAbs(operand);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FACGE

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

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

Scalar half precision

```
FACGE <Hd>, <Hn>, <Hm>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Scalar single-precision and double-precision

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

Scalar single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Vector half precision

(ARMv8.2)

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

Vector half precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Vector single-precision and double-precision

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

Vector single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Assembler Symbols

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

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

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

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

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

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

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

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

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

Operation

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

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FACGT

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

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

Scalar half precision

```
FACGT <Hd>, <Hn>, <Hm>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Scalar single-precision and double-precision

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

Scalar single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Vector half precision

(ARMv8.2)

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

Vector half precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Vector single-precision and double-precision

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

Vector single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Assembler Symbols

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

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

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

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

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

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

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

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

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

Operation

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

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADD (vector)

Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

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

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Single-precision and double-precision

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

Single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Assembler Symbols

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

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

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

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

Operation

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

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	Rm					0	0	1	0	1	0	Rn					Rd				
																			op												

Half-precision (type == 11)

(ARMv8.2)

```
FADD <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FADD <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FADD <Dd>, <Dn>, <Dm>
```

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

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPAdd(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADDP (scalar)

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

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

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

Half-precision

```
FADDP <V><d>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = 32;
```

Single-precision and double-precision

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

Single-precision and double-precision

```
FADDP <V><d>, <Vn>.<T>
```

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

integer esize = 32;
integer datasize = 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

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

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

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

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

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADDP (vector)

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

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

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

Half-precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Single-precision and double-precision

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

Single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>	For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

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

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

Operation

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

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCADD

Floating-point Complex Add.

This instruction adds corresponding complex numbers from the two source vector registers and writes the resulting complex numbers into the destination vector register. The number of complex numbers that can be stored in the source and the destination vector registers is calculated as the vector register size divided by the length of each complex number. Each complex number is represented in a SIMD&FP register as a pair of elements with the imaginary part of the number being placed in the more significant element, and the real part of the number being placed in the less significant element. Both real and imaginary parts of the source and the resulting complex number are represented as floating-point values.

One of the two vector elements that are read from each of the numbers in the second source SIMD&FP register can be optionally negated based on the rotation value:

- If the rotation is 90, the odd-numbered vector elements are negated.
- If the rotation is 270, the even-numbered vector elements are negated.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

Three registers of the same type

(ARMv8.3)

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

Three registers of the same type

```
FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>
```

```
if !HaveFCADDExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then ReservedValue();
if Q == '0' && size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then ReservedValue();
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

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

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

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

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

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

<rotate> Is the rotation, encoded in “rot”:

rot	<rotate>
0	90
1	270

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '0'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element3 = Elem[operand2, e*2, esize];
        when '1'
            element1 = Elem[operand2, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
    Elem[result, e*2, esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR);
    Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCCMP

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the *PSTATE*.{N, Z, C, V} flags. If the condition does not pass then the *PSTATE*.{N, Z, C, V} flags are set to the flag bit specifier.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

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

Half-precision (type == 11)

(ARMv8.2)

```
FCCMP <Hn>, <Hm>, #<nzcv>, <cond>
```

Single-precision (type == 00)

```
FCCMP <Sn>, <Sm>, #<nzcv>, <cond>
```

Double-precision (type == 01)

```
FCCMP <Dn>, <Dm>, #<nzcv>, <cond>
```

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

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

bits(4) flags = nzcv;
```

Assembler Symbols

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

Operation

```
CheckFPAdvSIMDEnabled64();

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

operand2 = V[m];

if ConditionHolds(cond) then
    flags = FPCompare(operand1, operand2, FALSE, FPCR);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

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

Half-precision (type == 11)

(ARMv8.2)

```
FCCMPE <Hn>, <Hm>, #<nzcv>, <cond>
```

Single-precision (type == 00)

```
FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>
```

Double-precision (type == 01)

```
FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>
```

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

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

bits(4) flags = nzcv;
```

Assembler Symbols

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

FCCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation

```
CheckFPAdvSIMDEnabled64();

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

operand2 = V[m];

if ConditionHolds(cond) then
    flags = FPCompare(operand1, operand2, TRUE, FPCR);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMEQ (register)

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

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

Scalar half precision

```
FCMEQ <Hd>, <Hn>, <Hm>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Scalar single-precision and double-precision

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

Scalar single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Vector half precision

(ARMv8.2)

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

Vector half precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Vector single-precision and double-precision

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

Vector single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Assembler Symbols

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

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

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

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

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

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

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

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

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

Operation

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

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMEQ (zero)

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

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

Scalar half precision

```
FCMEQ <Hd>, <Hn>, #0.0
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

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

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

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

Scalar single-precision and double-precision

```
FCMEQ <V><d>, <V><n>, #0.0
```

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

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision

(ARMv8.2)

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

Vector half precision

```
FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

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

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision

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

Vector single-precision and double-precision

```
FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0
```

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

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

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

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

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

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

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

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

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

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMGE (register)

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

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

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

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

Scalar half precision

```
FCMGE <Hd>, <Hn>, <Hm>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Scalar single-precision and double-precision

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

Scalar single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Vector half precision

(ARMv8.2)

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

Vector half precision

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

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Vector single-precision and double-precision

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

Vector single-precision and double-precision

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMGE (zero)

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U																	op												

Scalar half precision

```
FCMGE <Hd>, <Hn>, #0.0
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U																	op												

Scalar single-precision and double-precision

```
FCMGE <V><d>, <V><n>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U																	op												

Vector half precision

```
FCMGE <Vd>.<T>, <Vn>.<T>, #0.0
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U																	op												

Vector single-precision and double-precision

```
FCMGE <Vd>.<T>, <Vn>.<T>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMGT (register)

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	0	Rm					0	0	1	0	0	1	Rn					Rd				
		U						E												ac											

Scalar half precision

```
FCMGT <Hd>, <Hn>, <Hm>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	Rm					1	1	1	0	0	1	Rn					Rd				
		U						E												ac											

Scalar single-precision and double-precision

```
FCMGT <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	0	Rm					0	0	1	0	0	1	Rn					Rd				
		U						E												ac											

Vector half precision

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm					1	1	1	0	0	1	Rn					Rd				
		U						E												ac											

Vector single-precision and double-precision

```
FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UnallocatedEncoding();
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, FPCR);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMGT (zero)

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U																	op												

Scalar half precision

```
FCMGT <Hd>, <Hn>, #0.0
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U																	op												

Scalar single-precision and double-precision

```
FCMGT <V><d>, <V><n>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U																	op												

Vector half precision

```
FCMGT <Vd>.<T>, <Vn>.<T>, #0.0
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
		U																	op												

Vector single-precision and double-precision

```
FCMGT <Vd>.<T>, <Vn>.<T>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMLA (by element)

Floating-point Complex Multiply Accumulate (by element).

This instruction multiplies the complex numbers in the first source vector register by the specified complex number in the second source vector register, and adds the results to the corresponding complex numbers in the destination vector register. The number of complex numbers that can be stored in the source and the destination vector registers is calculated as the vector register size divided by the length of each complex number. Each complex number is represented in a SIMD&FP register as a pair of elements with the imaginary part of the number being placed in the more significant element, and the real part of the number being placed in the less significant element. Both real and imaginary parts of the source and the resulting complex number are represented as floating-point values.

None, one, or both of the two vector elements that are read from each of the numbers in the second source SIMD&FP register can be negated based on the rotation value:

- If the rotation is 0, none of the vector elements are negated.
- If the rotation is 90, the odd-numbered vector elements are negated.
- If the rotation is 180, both vector elements are negated.
- If the rotation is 270, the even-numbered vector elements are negated.

The indexed element variant of this instruction is available for half-precision and single-precision number values. For this variant, the index value determines the position in the specified element of the second source vector register of the single source value that is multiplied with each of the complex numbers in the first source vector register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				0	rot		1	H	0	Rn					Rd				

(size == 01)

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

(size == 10)

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

```
if !HaveFCADDExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
if size == '00' || size == '11' then ReservedValue();
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then ReservedValue();
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then ReservedValue();
if size == '01' && H == '1' && Q == '0' then ReservedValue();
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:H:L”:

size	<index>
00	RESERVED
01	H:L
10	H
11	RESERVED

<rotate> Is the rotation, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[m];
bits(datasize) operand2 = V[n];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '00'
            element1 = Elem[operand1, index*2, esize];
            element2 = Elem[operand2, e*2, esize];
            element3 = Elem[operand1, index*2+1, esize];
            element4 = Elem[operand2, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand1, index*2+1, esize]);
            element2 = Elem[operand2, e*2+1, esize];
            element3 = Elem[operand1, index*2, esize];
            element4 = Elem[operand2, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand1, index*2, esize]);
            element2 = Elem[operand2, e*2, esize];
            element3 = FPNeg(Elem[operand1, index*2+1, esize]);
            element4 = Elem[operand2, e*2, esize];
        when '11'
            element1 = Elem[operand1, index*2+1, esize];
            element2 = Elem[operand2, e*2+1, esize];
            element3 = FPNeg(Elem[operand1, index*2, esize]);
            element4 = Elem[operand2, e*2+1, esize];

    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, FPCR);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMLA

Floating-point Complex Multiply Accumulate.

This instruction multiplies corresponding complex numbers from the two source vector registers and adds the results to the corresponding complex numbers in the destination vector register. The number of complex numbers that can be stored in the source and the destination vector registers is calculated as the vector register size divided by the length of each complex number. Each complex number is represented in a SIMD&FP register as a pair of elements with the imaginary part of the number being placed in the more significant element, and the real part of the number being placed in the less significant element. Both real and imaginary parts of the source and the resulting complex number are represented as floating-point values.

None, one, or both of the two vector elements that are read from each of the numbers in the second source SIMD&FP register can be negated based on the rotation value:

- If the rotation is 0, none of the vector elements are negated.
- If the rotation is 90, the odd-numbered vector elements are negated.
- If the rotation is 180, both vector elements are negated.
- If the rotation is 270, the even-numbered vector elements are negated.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		0	Rm					1	1	0	rot		1	Rn					Rd				

Three registers of the same type

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>
```

```
if !HaveFCADDExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then ReservedValue();
if Q == '0' && size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then ReservedValue();
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate> Is the rotation, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '00'
            element1 = Elem[operand2, e*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, e*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, e*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, e*2, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, e*2+1, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, e*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];

    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, FPCR);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMLE (zero)

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn					Rd				
		U																	op												

Scalar half precision

```
FCMLE <Hd>, <Hn>, #0.0
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn					Rd				
		U																	op												

Scalar single-precision and double-precision

```
FCMLE <V><d>, <V><n>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector half precision
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn					Rd				
		U																	op												

Vector half precision

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn					Rd				
		U																	op												

Vector single-precision and double-precision

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMLT (zero)

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn					Rd				

Scalar half precision

```
FCMLT <Hd>, <Hn>, #0.0
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	0	1	0	Rn					Rd				

Scalar single-precision and double-precision

```
FCMLT <V><d>, <V><n>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn					Rd				

Vector half precision

```
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	0	1	0	Rn					Rd				

Vector single-precision and double-precision

```
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMP

Floating-point quiet Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the *PSTATE*.{N, Z, C, V} flags.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	Rm					0	0	1	0	0	0	Rn					0	x	0	0	0
																											opc				

Half-precision (type == 11 && opc == 00)

(ARMv8.2)

```
FCMP <Hn>, <Hm>
```

Half-precision, zero (type == 11 && Rm == (00000) && opc == 01)

(ARMv8.2)

```
FCMP <Hn>, #0.0
```

Single-precision (type == 00 && opc == 00)

```
FCMP <Sn>, <Sm>
```

Single-precision, zero (type == 00 && Rm == (00000) && opc == 01)

```
FCMP <Sn>, #0.0
```

Double-precision (type == 01 && opc == 00)

```
FCMP <Dn>, <Dm>
```

Double-precision, zero (type == 01 && Rm == (00000) && opc == 01)

```
FCMP <Dn>, #0.0
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);     // ignored when opc<0> == '1'

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

Assembler Symbols

<Dn>	For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Hn> For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sn> For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = if cmp_with_zero then FPZero('0') else V[m];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMPE

Floating-point signaling Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the *PSTATE*.{N, Z, C, V} flags.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	Rm					0	0	1	0	0	0	Rn					1	x	0	0	0
																											opc				

Half-precision (type == 11 && opc == 10)

(ARMv8.2)

```
FCMPE <Hn>, <Hm>
```

Half-precision, zero (type == 11 && Rm == (00000) && opc == 11)

(ARMv8.2)

```
FCMPE <Hn>, #0.0
```

Single-precision (type == 00 && opc == 10)

```
FCMPE <Sn>, <Sm>
```

Single-precision, zero (type == 00 && Rm == (00000) && opc == 11)

```
FCMPE <Sn>, #0.0
```

Double-precision (type == 01 && opc == 10)

```
FCMPE <Dn>, <Dm>
```

Double-precision, zero (type == 01 && Rm == (00000) && opc == 11)

```
FCMPE <Dn>, #0.0
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);     // ignored when opc<0> == '1'

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

Assembler Symbols

<Dn>	For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Hn> For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sn> For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

FCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = if cmp_with_zero then FPZero('0') else V[m];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	Rm					cond				1	1	Rn					Rd				

Half-precision (type == 11)

(ARMv8.2)

```
FCSEL <Hd>, <Hn>, <Hm>, <cond>
```

Single-precision (type == 00)

```
FCSEL <Sd>, <Sn>, <Sm>, <cond>
```

Double-precision (type == 01)

```
FCSEL <Dd>, <Dn>, <Dm>, <cond>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

bits(4) condition = cond;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;

result = if ConditionHolds(condition) then V[n] else V[m];

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVT

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the *FPCR* and writes the result to the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	0	1	opc		1	0	0	0	0	Rn					Rd				

Half-precision to single-precision (type == 11 && opc == 00)

```
FCVT <Sd>, <Hn>
```

Half-precision to double-precision (type == 11 && opc == 01)

```
FCVT <Dd>, <Hn>
```

Single-precision to half-precision (type == 00 && opc == 11)

```
FCVT <Hd>, <Sn>
```

Single-precision to double-precision (type == 00 && opc == 01)

```
FCVT <Dd>, <Sn>
```

Double-precision to half-precision (type == 01 && opc == 11)

```
FCVT <Hd>, <Dn>
```

Double-precision to single-precision (type == 01 && opc == 00)

```
FCVT <Sd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if type == opc then UnallocatedEncoding();

integer srcsize;
case type of
    when '00' srcsize = 32;
    when '01' srcsize = 64;
    when '10' UnallocatedEncoding();
    when '11' srcsize = 16;
integer dstsize;
case opc of
    when '00' dstsize = 32;
    when '01' dstsize = 64;
    when '10' UnallocatedEncoding();
    when '11' dstsize = 16;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(dstsize) result;
bits(srcsize) operand = V[n];

result = FPConvert(operand, FPCR);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTAS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn					Rd				
		U																													

Scalar half precision

```
FCVTAS <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn					Rd				
		U																													

Scalar single-precision and double-precision

```
FCVTAS <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn					Rd				
		U																													

Vector half precision

```
FCVTAS <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn					Rd				
		U																													

Vector single-precision and double-precision

```
FCVTAS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	0	0	1	0	0	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTAS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTAS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTAS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTAS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTAS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTAS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, FPRounding_TIEAWAY);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTAU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn					Rd				
		U																													

Scalar half precision

```
FCVTAU <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn					Rd				
		U																													

Scalar single-precision and double-precision

```
FCVTAU <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn					Rd				
		U																													

Vector half precision

```
FCVTAU <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn					Rd				
		U																													

Vector single-precision and double-precision

```
FCVTAU <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	0	0	1	0	1	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTAU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTAU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTAU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTAU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTAU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTAU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, FPRounding_TIEAWAY);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTL, FCVTL2

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the *FPCR*, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the `FCVTL2` variant operates on the elements in the top 64 bits of the source 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	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	1	1	0	Rn					Rd				

Vector single-precision and double-precision

```
FCVTL{2} <Vd>.<Ta>, <Vn>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
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 “sz”:

sz	<Ta>
0	4S
1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	0	4H
0	1	8H
1	0	2S
1	1	4S

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;

for e = 0 to elements-1
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR);

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTMS (vector)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Scalar half precision

```
FCVTMS <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Scalar single-precision and double-precision

```
FCVTMS <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Vector half precision

```
FCVTMS <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Vector single-precision and double-precision

```
FCVTMS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	1	0	0	0	0	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTMS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTMS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTMS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTMS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTMS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTMS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTMU (vector)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Scalar half precision

```
FCVTMU <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Scalar single-precision and double-precision

```
FCVTMU <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Vector half precision

```
FCVTMU <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Vector single-precision and double-precision

```
FCVTMU <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	1	0	0	0	1	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTMU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTMU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTMU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTMU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTMU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTMU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTN, FCVTN2

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the *FPCR*.

The `FCVTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `FCVTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the 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
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn					Rd				

Vector single-precision and double-precision

```
FCVTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	0	4H
0	1	8H
1	0	2S
1	1	4S

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	4S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR);

Vpart[d, part] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTNS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Scalar half precision

```
FCVTNS <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Scalar single-precision and double-precision

```
FCVTNS <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Vector half precision

```
FCVTNS <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Vector single-precision and double-precision

```
FCVTNS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	0	0	0	0	0	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTNS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTNS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTNS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTNS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTNS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTNS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTNU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Scalar half precision

```
FCVTNU <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Scalar single-precision and double-precision

```
FCVTNU <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Vector half precision

```
FCVTNU <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Vector single-precision and double-precision

```
FCVTNU <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	0	0	0	0	1	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTNU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTNU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTNU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTNU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTNU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTNU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTPS (vector)

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Scalar half precision

```
FCVTPS <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Scalar single-precision and double-precision

```
FCVTPS <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Vector half precision

```
FCVTPS <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Vector single-precision and double-precision

```
FCVTPS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	0	1	0	0	0	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTPS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTPS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTPS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTPS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTPS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTPS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTPU (vector)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Scalar half precision

```
FCVTPU <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Scalar single-precision and double-precision

```
FCVTPU <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Vector half precision

```
FCVTPU <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U						o2											o1												

Vector single-precision and double-precision

```
FCVTPU <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	0	1	0	0	1	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTPU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTPU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTPU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTPU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTPU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTPU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTXN, FCVTXN2

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD&FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The `FCVTXN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `FCVTXN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn					Rd				

Scalar

```
FCVTXN <Vb><d>, <Va><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then ReservedValue();
integer esize = 32;
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn					Rd				

Vector

```
FCVTXN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then ReservedValue();
integer esize = 32;
integer datasize = 64;
integer elements = 2;
integer part = UInt(Q);
```

Assembler Symbols

2	Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	x	RESERVED
1	0	2S
1	1	4S

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	RESERVED
1	2D

<Vb> Is the destination width specifier, encoded in “sz”:

sz	<Vb>
0	RESERVED
1	S

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “sz”:

sz	<Va>
0	RESERVED
1	D

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR, FPRounding_ODD);

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTZS (vector, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	Rn					Rd				
		U							immh																						

Scalar

```
FCVTZS <V><d>, <V><n>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	Rn					Rd				
		U							immh																						

Vector

```
FCVTZS <Vd>.<T>, <Vn>.<T>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<V>	Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTZS (vector, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Scalar half precision

```
FCVTZS <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Scalar single-precision and double-precision

```
FCVTZS <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Vector half precision

```
FCVTZS <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Vector single-precision and double-precision

```
FCVTZS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type		0	1	1	0	0	0	scale						Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTZS <Wd>, <Hn>, #<fbits>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTZS <Xd>, <Hn>, #<fbits>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTZS <Wd>, <Sn>, #<fbits>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTZS <Xd>, <Sn>, #<fbits>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTZS <Wd>, <Dn>, #<fbits>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTZS <Xd>, <Dn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale". For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, fracbits, FALSE, FPCR, FPRounding_ZERO);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	1	1	0	0	0	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTZS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTZS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTZS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTZS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTZS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTZS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTZU (vector, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	Rn					Rd				
		U							immh																						

Scalar

```
FCVTZU <V><d>, <V><n>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	Rn					Rd				
		U							immh																						

Vector

```
FCVTZU <Vd>.<T>, <Vn>.<T>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<V>	Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTZU (vector, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Scalar half precision

```
FCVTZU <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Scalar single-precision and double-precision

```
FCVTZU <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Vector half precision

```
FCVTZU <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn					Rd				
		U						o2											o1												

Vector single-precision and double-precision

```
FCVTZU <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type		0	1	1	0	0	1	scale						Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTZU <Wd>, <Hn>, #<fbits>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTZU <Xd>, <Hn>, #<fbits>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTZU <Wd>, <Sn>, #<fbits>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTZU <Xd>, <Sn>, #<fbits>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTZU <Wd>, <Dn>, #<fbits>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTZU <Xd>, <Dn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale". For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, fracbits, TRUE, FPCR, FPRounding_ZERO);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	1	1	0	0	1	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11)
(ARMv8.2)

```
FCVTZU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11)
(ARMv8.2)

```
FCVTZU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && type == 00)

```
FCVTZU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && type == 00)

```
FCVTZU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && type == 01)

```
FCVTZU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && type == 01)

```
FCVTZU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = intval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FDIV (vector)

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm					0	0	1	1	1	1	Rn					Rd				

Half-precision

```
FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm					1	1	1	1	1	1	Rn					Rd				

Single-precision and double-precision

```
FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FDIV (scalar)

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	Rm					0	0	0	1	1	0	Rn					Rd				

Half-precision (type == 11)

(ARMv8.2)

```
FDIV <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FDIV <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FDIV <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPDiv(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FJCVTZS

Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be accommodated as a signed 32-bit integer, then the result is the integer modulo 2^{32}, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Double-precision to 32-bit

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	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	1	1	1	1	1	1	0	0	0	0	0	0	0	Rn					Rd				
sf								type			rmode		opcode																		

Double-precision to 32-bit

```
FJCVTZS <Wd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HaveFJCVTZSExt() then UnallocatedEncoding();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) fltval;
bits(32) intval;

fltval = V[n];
intval = FPToFixedJS(fltval, FPCR, TRUE);
X[d] = ZeroExtend(intval<31:0>, 64);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMADD

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	type		0	Rm					0	Ra					Rn					Rd				
										o1						o0															

Half-precision (type == 11)

(ARMv8.2)

```
FMADD <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (type == 00)

```
FMADD <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (type == 01)

```
FMADD <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAX (vector)

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm					0	0	1	1	0	1	Rn					Rd				
		U						o1																							

Half-precision

```
FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm					1	1	1	1	0	1	Rn					Rd				
		U						o1																							

Single-precision and double-precision

```
FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>	For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAX (scalar)

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	Rm					0	1	0	0	1	0	Rn					Rd				
																		op													

Half-precision (type == 11)

(ARMv8.2)

```
FMAX <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FMAX <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FMAX <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMax(operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXNM (vector)

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm					0	0	0	0	0	1	Rn					Rd				
		U						a																							

Half-precision

```
FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm					1	1	0	0	0	1	Rn					Rd				
		U						o1																							

Single-precision and double-precision

```
FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>	For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	Rm					0	1	1	0	1	0	Rn					Rd				
																		op													

Half-precision (type == 11)

(ARMv8.2)

```
FMAXNM <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FMAXNM <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FMAXNM <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMaxNum(operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1	sz																						

Half-precision

```
FMAXNMP <V><d>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

```
FMAXNMP <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAXNUM, operand, esize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm					0	0	0	0	0	1	Rn					Rd				
		U						a																							

Half-precision

```
FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm					1	1	0	0	0	1	Rn					Rd				
		U						o1																							

Single-precision and double-precision

```
FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1																							

Half-precision

```
FMAXNMV <V><d>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

```
FMAXNMV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then ReservedValue();     // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAXNUM, operand, esize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1	sz																						

Half-precision

```
FMAXP <V><d>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

```
FMAXP <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAX, operand, esize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXP (vector)

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm					0	0	1	1	0	1	Rn					Rd				
		U						o1																							

Half-precision

```
FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm					1	1	1	1	0	1	Rn					Rd				
		U						o1																							

Single-precision and double-precision

```
FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

Half-precision

```
FMAXV <V><d>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

```
FMAXV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then ReservedValue();

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAX, operand, esize);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMIN (vector)

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm					0	0	1	1	0	1	Rn					Rd				
		U						o1																							

Half-precision

```
FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm					1	1	1	1	0	1	Rn					Rd				
		U						o1																							

Single-precision and double-precision

```
FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>	For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	Rm					0	1	0	1	1	0	Rn					Rd				
																		op													

Half-precision (type == 11)

(ARMv8.2)

```
FMIN <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FMIN <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FMIN <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMin(operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINNM (vector)

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm					0	0	0	0	0	1	Rn					Rd				
		U						a																							

Half-precision

```
FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm					1	1	0	0	0	1	Rn					Rd				
		U						o1																							

Single-precision and double-precision

```
FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>	For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	Rm					0	1	1	1	1	0	Rn					Rd				
																		op													

Half-precision (type == 11)

(ARMv8.2)

```
FMINNM <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FMINNM <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FMINNM <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMinNum(operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1	sz																						

Half-precision

```
FMINNMP <V><d>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

```
FMINNMP <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMINNUM, operand, esize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	0	Rm					0	0	0	0	0	1	Rn					Rd				
		U						a																							

Half-precision

```
FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm					1	1	0	0	0	1	Rn					Rd				
		U						o1																							

Single-precision and double-precision

```
FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINNMV

Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1																							

Half-precision

```
FMINNMV <V><d>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

```
FMINNMV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then ReservedValue();     // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMINNUM, operand, esize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1	sz																						

Half-precision

```
FMINP <V><d>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

```
FMINP <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMIN, operand, esize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINP (vector)

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	0	Rm					0	0	1	1	0	1	Rn					Rd				
		U						o1																							

Half-precision

```
FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm					1	1	1	1	0	1	Rn					Rd				
		U						o1																							

Single-precision and double-precision

```
FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINV

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

Half-precision

```
FMINV <V><d>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

Single-precision and double-precision

```
FMINV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then ReservedValue();

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMIN, operand, esize);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLA (by element)

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	0	L	M	Rm				0	0	0	1	H	0	Rn					Rd				
																	o2														

Scalar, half-precision

```
FMLA <Hd>, <Hn>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm				0	0	0	1	H	0	Rn					Rd				
																	o2														

Scalar, single-precision and double-precision

```
FMLA <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Vector, half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	L	M	Rm				0	0	0	1	H	0	Rn					Rd				
																	o2														

Vector, half-precision

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Vector, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	1	sz	L	M	Rm				0	0	0	1	H	0	Rn					Rd				
																	o2														

Vector, single-precision and double-precision

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLA (vector)

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm					0	0	0	0	1	1	Rn					Rd				
								a																							

Half-precision

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm					1	1	0	0	1	1	Rn					Rd				
								op																							

Single-precision and double-precision

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLS (by element)

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	0	L	M	Rm				0	1	0	1	H	0	Rn					Rd				
																	o2														

Scalar, half-precision

```
FMLS <Hd>, <Hn>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm				0	1	0	1	H	0	Rn					Rd				
																	o2														

Scalar, single-precision and double-precision

```
FMLS <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Vector, half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	L	M	Rm				0	1	0	1	H	0	Rn					Rd				
																	o2														

Vector, half-precision

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Vector, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	1	sz	L	M	Rm				0	1	0	1	H	0	Rn					Rd				
																	o2														

Vector, single-precision and double-precision

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLS (vector)

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm					0	0	0	0	1	1	Rn					Rd				
								a																							

Half-precision

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm					1	1	0	0	1	1	Rn					Rd				
								op																							

Single-precision and double-precision

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	1	1	d	e	f	g	h	Rd				

Half-precision

```
FMOV <Vd>.<T>, #<imm>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;

imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:Zeros(6);

imm = Replicate(imm16, datasize DIV 16);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	0	1	d	e	f	g	h	Rd				
																cmode															

Single-precision (op == 0)

```
FMOV <Vd>.<T>, #<imm>
```

Double-precision (Q == 1 && op == 1)

```
FMOV <Vd>.2D, #<imm>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

if cmode:op == '11111' then
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UnallocatedEncoding();

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in A64 floating-point instructions*.

Operation

```
CheckFPAdvSIMDEnabled64();

V[rd] = imm;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	0	0	0	0	1	0	0	0	0	Rn					Rd				
															opc																

Half-precision (type == 11)

(ARMv8.2)

```
FMOV <Hd>, <Hn>
```

Single-precision (type == 00)

```
FMOV <Sd>, <Sn>
```

Double-precision (type == 01)

```
FMOV <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

V[d] = operand;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type		1	0	x	1	1	x	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

Half-precision to 32-bit (sf == 0 && type == 11 && rmode == 00 && opcode == 110)
(ARMv8.2)

```
FMOV <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && type == 11 && rmode == 00 && opcode == 110)
(ARMv8.2)

```
FMOV <Xd>, <Hn>
```

32-bit to half-precision (sf == 0 && type == 11 && rmode == 00 && opcode == 111)
(ARMv8.2)

```
FMOV <Hd>, <Wn>
```

32-bit to single-precision (sf == 0 && type == 00 && rmode == 00 && opcode == 111)

```
FMOV <Sd>, <Wn>
```

Single-precision to 32-bit (sf == 0 && type == 00 && rmode == 00 && opcode == 110)

```
FMOV <Wd>, <Sn>
```

64-bit to half-precision (sf == 1 && type == 11 && rmode == 00 && opcode == 111)
(ARMv8.2)

```
FMOV <Hd>, <Xn>
```

64-bit to double-precision (sf == 1 && type == 01 && rmode == 00 && opcode == 111)

```
FMOV <Dd>, <Xn>
```

64-bit to top half of 128-bit (sf == 1 && type == 10 && rmode == 01 && opcode == 111)

```
FMOV <Vd>.D[1], <Xn>
```

Double-precision to 64-bit (sf == 1 && type == 01 && rmode == 00 && opcode == 110)

```
FMOV <Xd>, <Dn>
```

Top half of 128-bit to 64-bit (sf == 1 && type == 10 && rmode == 01 && opcode == 110)

```
FMOV <Xd>, <Vn>.D[1]
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UnallocatedEncoding();
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

case opcode<2:1>:rmode of
    when '00 xx'    // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'    // [US]CVTF
        rounding = FPRoundingMode(FPCR);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'    // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'    // FMOV
        if fltsize != 16 && fltsize != intsize then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'    // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UnallocatedEncoding();
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
        fltsize = 64;    // size of D[1] is 64
    when '11 11'    // FJCVTZS
        if !HaveFJCVTZSExt() then UnallocatedEncoding();
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n];
        intval = FPToFixed(fltval, 0, unsigned, FPCR, rounding);
        X[d] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n];
        fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
        V[d] = fltval;
    when FPConvOp_MOV_FtoI
        fltval = Vpart[n, part];
        intval = ZeroExtend(fltval, intsize);
        X[d] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n];
        fltval = intval<fltsize-1:0>;
        Vpart[d, part] = fltval;
    when FPConvOp_CVT_FtoI_JS
        fltval = V[n];
        intval = FPToFixedJS(fltval, FPCR, TRUE);
        X[d] = ZeroExtend(intval<31:0>, 64);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMOV (scalar, immediate)

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD&FP destination register. Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	imm8								1	0	0	0	0	0	0	0	Rd				

Half-precision (type == 11)
(ARMv8.2)

```
FMOV <Hd>, #<imm>
```

Single-precision (type == 00)

```
FMOV <Sd>, #<imm>
```

Double-precision (type == 01)

```
FMOV <Dd>, #<imm>
```

```
integer d = UInt(Rd);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

bits(datasize) imm = VFPExpandImm(imm8);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<imm>	Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in A64 floating-point instructions*.

Operation

```
CheckFPAdvSIMDEnabled64();

V[d] = imm;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	type		0	Rm					1	Ra					Rn					Rd				
										o1						o0															

Half-precision (type == 11)

(ARMv8.2)

```
FMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (type == 00)

```
FMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (type == 01)

```
FMSUB <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMUL (by element)

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	0	L	M	Rm				1	0	0	1	H	0	Rn					Rd				
		U																													

Scalar, half-precision

```
FMUL <Hd>, <Hn>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm				1	0	0	1	H	0	Rn					Rd				
		U																													

Scalar, single-precision and double-precision

```
FMUL <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Vector, half-precision
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	L	M	Rm				1	0	0	1	H	0	Rn					Rd				
		U																													

Vector, half-precision

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	1	sz	L	M	Rm				1	0	0	1	H	0	Rn					Rd				
		U																													

Vector, single-precision and double-precision

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMUL (vector)

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm					0	0	0	1	1	1	Rn					Rd				

Half-precision

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm					1	1	0	1	1	1	Rn					Rd				

Single-precision and double-precision

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMUL (scalar)

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	Rm					0	0	0	0	1	0	Rn					Rd				
																op															

Half-precision (type == 11)

(ARMv8.2)

```
FMUL <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FMUL <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FMUL <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMULX (by element)

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	0	L	M	Rm				1	0	0	1	H	0	Rn					Rd				
		U																													

Scalar, half-precision

```
FMULX <Hd>, <Hn>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	sz	L	M	Rm				1	0	0	1	H	0	Rn					Rd				
		U																													

Scalar, single-precision and double-precision

```
FMULX <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Vector, half-precision
(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	0	L	M	Rm				1	0	0	1	H	0	Rn					Rd				
		U																													

Vector, half-precision

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
index = UInt(H:L:M);

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	1	sz	L	M	Rm				1	0	0	1	H	0	Rn					Rd				
		U																													

Vector, single-precision and double-precision

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector, half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector, single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMULX

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	0	1	1	1	1	0	0	1	0	Rm	0	0	0	1	1	1	Rn	Rd

Scalar half precision

```
FMULX <Hd>, <Hn>, <Hm>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	0	1	1	1	1	0	0	sz	1	Rm	1	1	0	1	1	1	Rn	Rd

Scalar single-precision and double-precision

```
FMULX <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	0	1	0	Rm	0	0	0	1	1	1	Rn	Rd

Vector half precision

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	0	sz	1	Rm	1	1	0	1	1	1	Rn	Rd

Vector single-precision and double-precision

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNEG (vector)

Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	1	1	0	Rn					Rd				
		U																													

Half-precision

```
FNEG <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
		U																													

Single-precision and double-precision

```
FNEG <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	0	0	1	0	1	0	0	0	0	Rn					Rd				
															opc																

Half-precision (type == 11)

(ARMv8.2)

```
FNEG <Hd>, <Hn>
```

Single-precision (type == 00)

```
FNEG <Sd>, <Sn>
```

Double-precision (type == 01)

```
FNEG <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPNeg(operand);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNMADD

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	type		1	Rm					0	Ra					Rn					Rd				
										o1						o0															

Half-precision (type == 11)

(ARMv8.2)

```
FNMADD <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (type == 00)

```
FNMADD <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (type == 01)

```
FNMADD <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operanda = FPNeg(operanda);
operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNMSUB

Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	type		1	Rm					1	Ra					Rn					Rd				
										o1						o0															

Half-precision (type == 11)

(ARMv8.2)

```
FNMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (type == 00)

```
FNMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (type == 01)

```
FNMSUB <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operanda = FPNeg(operanda);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	Rm					1	0	0	0	1	0	Rn					Rd				
																op															

Half-precision (type == 11)

(ARMv8.2)

```
FNMUL <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FNMUL <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FNMUL <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);

result = FPNeg(result);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRECPE

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				

Scalar half precision

```
FRECPE <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				

Scalar single-precision and double-precision

```
FRECPE <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				

Vector half precision

```
FRECPE <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				

Vector single-precision and double-precision

```
FRECPE <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRECPS

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	0	1	1	1	1	0	0	1	0	Rm	0	0	1	1	1	1	Rn	Rd

Scalar half precision

```
FRECPS <Hd>, <Hn>, <Hm>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	0	1	1	1	1	0	0	sz	1	Rm	1	1	1	1	1	1	Rn	Rd

Scalar single-precision and double-precision

```
FRECPS <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	0	1	0	Rm	0	0	1	1	1	1	Rn	Rd

Vector half precision

```
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	0	sz	1	Rm	1	1	1	1	1	1	Rn	Rd

Vector single-precision and double-precision

```
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRECPX

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn					Rd				

Half-precision

```
FRECPX <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn					Rd				

Single-precision and double-precision

```
FRECPX <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecpX(element, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTA (vector)

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn					Rd				
		U						o2											o1												

Half-precision

```
FRINTA <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn					Rd				
		U						o2											o1												

Single-precision and double-precision

```
FRINTA <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	1	1	0	0	1	0	0	0	0	Rn					Rd				
														rmode																	

Half-precision (type == 11)

(ARMv8.2)

```
FRINTA <Hd>, <Hn>
```

Single-precision (type == 00)

```
FRINTA <Sd>, <Sn>
```

Double-precision (type == 01)

```
FRINTA <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, FPRounding_TIEAWAY, FALSE);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTI (vector)

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn					Rd				
		U						o2											o1												

Half-precision

```
FRINTI <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn					Rd				
		U						o2											o1												

Single-precision and double-precision

```
FRINTI <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	1	1	1	1	1	0	0	0	0	Rn					Rd				
														rmode																	

Half-precision (type == 11)

(ARMv8.2)

```
FRINTI <Hd>, <Hn>
```

Single-precision (type == 00)

```
FRINTI <Sd>, <Sn>
```

Double-precision (type == 01)

```
FRINTI <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

FPRounding rounding;
rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn					Rd				
		U						o2											o1												

Half-precision

```
FRINTM <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn					Rd				
		U						o2											o1												

Single-precision and double-precision

```
FRINTM <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	1	0	1	0	1	0	0	0	0	Rn					Rd				
														rmode																	

Half-precision (type == 11)

(ARMv8.2)

```
FRINTM <Hd>, <Hn>
```

Single-precision (type == 00)

```
FRINTM <Sd>, <Sn>
```

Double-precision (type == 01)

```
FRINTM <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

FPRounding rounding;
rounding = FPDecodeRounding('10');
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn					Rd				
		U						o2											o1												

Half-precision

```
FRINTN <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn					Rd				
		U						o2											o1												

Single-precision and double-precision

```
FRINTN <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	1	0	0	0	1	0	0	0	0	Rn					Rd				
														rmode																	

Half-precision (type == 11)

(ARMv8.2)

```
FRINTN <Hd>, <Hn>
```

Single-precision (type == 00)

```
FRINTN <Sd>, <Sn>
```

Double-precision (type == 01)

```
FRINTN <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

FPRounding rounding;
rounding = FPDecodeRounding('00');
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTP (vector)

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	0	1	0	Rn					Rd				
		U						o2											o1												

Half-precision

```
FRINTP <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	0	0	1	0	Rn					Rd				
		U						o2											o1												

Single-precision and double-precision

```
FRINTP <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	1	0	0	1	1	0	0	0	0	Rn					Rd				
														rmode																	

Half-precision (type == 11)

(ARMv8.2)

```
FRINTP <Hd>, <Hn>
```

Single-precision (type == 00)

```
FRINTP <Sd>, <Sn>
```

Double-precision (type == 01)

```
FRINTP <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

FPRounding rounding;
rounding = FPDecodeRounding('01');
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTX (vector)

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

An Inexact exception is raised when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn					Rd				
		U						o2											o1												

Half-precision

```
FRINTX <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn					Rd				
		U						o2											o1												

Single-precision and double-precision

```
FRINTX <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

An Inexact exception is raised when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	1	1	1	0	1	0	0	0	0	Rn					Rd				
														rmode																	

Half-precision (type == 11)

(ARMv8.2)

```
FRINTX <Hd>, <Hn>
```

Single-precision (type == 00)

```
FRINTX <Sd>, <Sn>
```

Double-precision (type == 01)

```
FRINTX <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

FPRounding rounding;
rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, TRUE);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTZ (vector)

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	0	1	1	0	Rn					Rd				
		U						o2											o1												

Half-precision

```
FRINTZ <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	0	1	1	0	Rn					Rd				
		U						o2											o1												

Single-precision and double-precision

```
FRINTZ <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UnallocatedEncoding();
    when '110' rounding = FPRoundingMode(FPCR); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	1	0	1	1	1	0	0	0	0	Rn					Rd				
														rmode																	

Half-precision (type == 11)

(ARMv8.2)

```
FRINTZ <Hd>, <Hn>
```

Single-precision (type == 00)

```
FRINTZ <Sd>, <Sn>
```

Double-precision (type == 01)

```
FRINTZ <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();

FPRounding rounding;
rounding = FPDecodeRounding('11');
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRSQRTE

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				

Scalar half precision

```
FRSQRTE <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				

Scalar single-precision and double-precision

```
FRSQRTE <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				

Vector half precision

```
FRSQRTE <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				

Vector single-precision and double-precision

```
FRSQRTE <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRSQRTS

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	0	1	1	1	1	0	1	1	0	Rm	0	0	1	1	1	1	Rn	Rd

Scalar half precision

```
FRSQRTS <Hd>, <Hn>, <Hm>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	0	1	1	1	1	0	1	sz	1	Rm	1	1	1	1	1	1	Rn	Rd

Scalar single-precision and double-precision

```
FRSQRTS <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	1	1	0	Rm	0	0	1	1	1	1	Rn	Rd

Vector half precision

```
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12 11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	1	sz	1	Rm	1 1 1 1 1	1	Rn	Rd

Vector single-precision and double-precision

```
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSQRT (vector)

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn					Rd				

Half-precision

```
FSQRT <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn					Rd				

Single-precision and double-precision

```
FSQRT <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPSqrt(element, FPCR);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	0	0	0	0	1	1	1	0	0	0	0	Rn					Rd				
															opc																

Half-precision (type == 11)

(ARMv8.2)

```
FSQRT <Hd>, <Hn>
```

Single-precision (type == 00)

```
FSQRT <Sd>, <Sn>
```

Double-precision (type == 01)

```
FSQRT <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPSqrt(operand, FPCR);

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSUB (vector)

Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm					0	0	0	1	0	1	Rn					Rd				
		U																													

Half-precision

```
FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm					1	1	0	1	0	1	Rn					Rd				
		U																													

Single-precision and double-precision

```
FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) diff;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	type		1	Rm					0	0	1	1	1	0	Rn					Rd				
																			op												

Half-precision (type == 11)

(ARMv8.2)

```
FSUB <Hd>, <Hn>, <Hm>
```

Single-precision (type == 00)

```
FSUB <Sd>, <Sn>, <Sm>
```

Double-precision (type == 01)

```
FSUB <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case type of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UnallocatedEncoding();
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPSub(operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

HINT

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are unallocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture.

31	30	29	28	27	26	25	24	23	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

Hints 6 and 7 (CRm == 0000 && op2 == 11x)

```
HINT #<imm>
```

Hints 8 to 15, and 24 to 127 (CRm != 00x0)

```
HINT #<imm>
```

Hints 17 to 23 (CRm == 0010 && op2 != 00x)

```
HINT #<imm>
```

Assembler Symbols

<imm> Is a 7-bit unsigned immediate, in the range 0 to 127, excluding the allocated encodings described below, encoded in "CRm:op2".

The following encodings of "CRm:op2" are allocated:

0000000

NOP

0000001

YIELD

0000010

WFE

0000011

WFI

0000100

SEV

0000101

SEVL

For allocated encodings of "CRm:op2":

- A disassembler will disassemble the allocated instruction, rather than the HINT instruction.
- An assembler may support assembly of allocated encodings using HINT with the corresponding <imm> value, but it is not required to do so.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

HLT

Halt instruction generates a Halt Instruction debug event.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	1	0	imm16																0	0	0	0	0

System

```
HLT #<imm>
```

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UndefinedFault();
```

Assembler Symbols

<imm>	Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
Halt(DebugHalt_HaltInstruction);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

HVC

Hypervisor Call causes an exception to EL2. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- At EL0, and Secure EL1.
- When *SCR_EL3*.HCE is set to 0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in *ESR_ELx*, using the EC value 0x16, and the value of the immediate argument.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	0	imm16																0	0	0	1	0

System

```
HVC #<imm>
```

```
bits(16) imm = imm16;
```

Assembler Symbols

<imm>	Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && IsSecure()) then
    UnallocatedEncoding();

hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);
if hvc_enable == '0' then
    AArch64.UndefinedFault();
else
    AArch64.CallHypervisor(imm);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

IC

Instruction Cache operation. For more information, see *A64 system instructions for cache maintenance*.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	1	op1			0	1	1	1	CRm				op2			Rt				
										L						CRn															

System

```
IC <ic_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1,'0111',CRm,op2) == Sys_IC`.

Assembler Symbols

<ic_op> Is an IC instruction name, as listed for the IC system instruction pages, encoded in “op1:CRm:op2”:

op1	CRm	op2	<ic_op>
000	0001	000	IALLUIS
000	0101	000	IALLU
011	0101	001	IVAU

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

INS (element)

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (element).

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12 11	10	9 8 7 6 5	4 3 2 1 0
0	1	1	0	1	1	1	0	0	0	0	imm5	0	imm4	1	Rn	Rd

Advanced SIMD

```
INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UnallocatedEncoding();

integer dst_index = UInt(imm5<4:size+1>);
integer src_index = UInt(imm4<3:size>);
integer idxdsize = if imm4<3> == '1' then 128 else 64;
// imm4<size-1:0> is IGNORED

integer esize = 8 << size;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index1> Is the destination element index encoded in “imm5”:

imm5	<index1>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index2> Is the source element index encoded in “imm5:imm4”:

imm5	<index2>
x0000	RESERVED
xxxx1	imm4<3:0>
xxx10	imm4<3:1>
xx100	imm4<3:2>
x1000	imm4<3>

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(128) result;

result = V[d];
Elem[result, dst_index, esize] = Elem[operand, src_index, esize];
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

INS (general)

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (from general).

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	0	0	1	1	1	0	0	0	0	imm5	0	0	0	1	1	1	Rn	Rd

Advanced SIMD

```
INS <Vd>.<Ts>[<index>], <R><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);

if size > 3 then UnallocatedEncoding();
integer index = UInt(imm5<4:size+1>);

integer esize = 8 << size;
integer datasize = 128;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(datasize) result;

result = V[d];
Elem[result, index, esize] = element;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm				1	1	0	1	1	1	1	1
																									opc						

System

```
ISB {<option>|#<imm>}
```

```
// Empty.
```

Assembler Symbols

<option> Specifies an optional limitation on the barrier operation. Values are:

SY

Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.

All other encodings of CRm are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

<imm> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

```
InstructionSynchronizationBarrier();
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	x	x	1	x	size		Rn					Rt				
									L							opcode															

One register (opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>]
```

Two registers (opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Three registers (opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Four registers (opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm					x	x	1	x	size		Rn					Rt				
									L							opcode															

One register, immediate offset (Rm == 11111 && opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	x	x	0	S	size		Rn					Rt				
									L	R						opcode															

8-bit (opcode == 000)

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					x	x	0	S	size		Rn					Rt				
									L	R						opcode															

8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	size		Rn					Rt				
									L	R						opcode			S											

No offset

```
LD1R { <Vt>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					1	1	0	0	size		Rn					Rt				
									L	R						opcode			S											

Immediate offset (Rm == 11111)

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#1
01	#2
10	#4
11	#8

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2 (multiple structures)

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see `LD3 (multiple structures)`.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	size		Rn					Rt				
									L							opcode															

No offset

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm					1	0	0	0	size		Rn					Rt				
									L							opcode															

Immediate offset (Rm == 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2 (single structure)

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	0	S	size		Rn					Rt				
									L	R						opcode															

8-bit (opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm					x	x	0	S	size		Rn					Rt				
									L	R						opcode															

8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2R

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	size		Rn					Rt				
									L	R						opcode			S											

No offset

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm					1	1	0	0	size		Rn					Rt				
									L	R						opcode			S											

Immediate offset (Rm == 11111)

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#2
01	#4
10	#8
11	#16

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

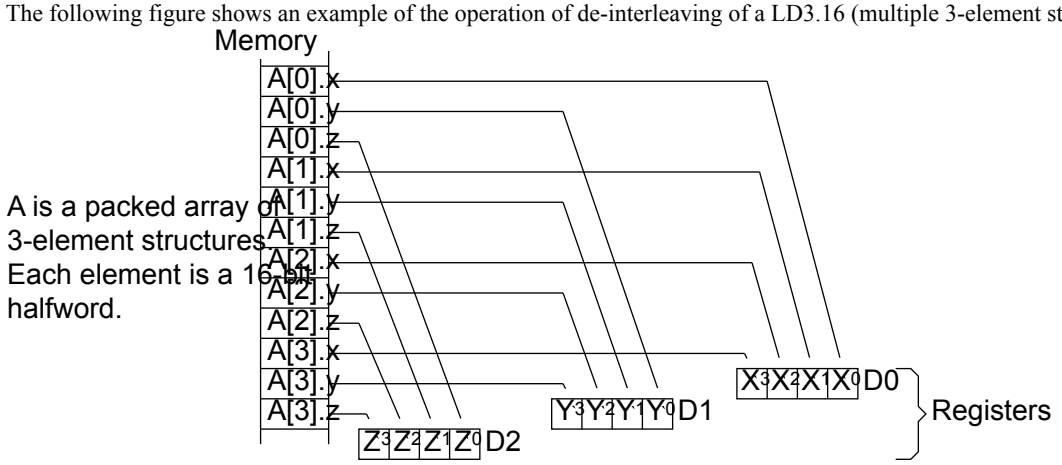
```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD3 (multiple structures)

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	size		Rn					Rt				
									L							opcode															

No offset

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm					0	1	0	0	size		Rn					Rt				
									L							opcode															

Immediate offset (Rm == 11111)

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD3 (single structure)

Load single 3-element structure to one lane of three registers). This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	x	x	1	S	size		Rn					Rt				
									L	R						opcode															

8-bit (opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					x	x	1	S	size		Rn					Rt				
									L	R						opcode															

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	1	1	1	0	size		Rn					Rt				
									L	R						opcode			S											

No offset

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm					1	1	1	0	size		Rn					Rt				
									L	R						opcode			S											

Immediate offset (Rm == 11111)

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#3
01	#6
10	#12
11	#24

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4 (multiple structures)

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see `LD3 (multiple structures)`.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	size		Rn					Rt				
									L							opcode															

No offset

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm					0	0	0	0	size		Rn					Rt				
									L							opcode															

Immediate offset (Rm == 11111)

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	1	S	size		Rn					Rt				
									L	R						opcode															

8-bit (opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm					x	x	1	S	size		Rn					Rt				
									L	R						opcode															

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	1	0	size		Rn					Rt				
									L	R						opcode			S											

No offset

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm					1	1	1	0	size		Rn					Rt				
									L	R						opcode			S											

Immediate offset (Rm == 11111)

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#4
01	#8
10	#16
11	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDADDA` and `LDADDAL` load from memory with acquire semantics.
- `LDADDL` and `LDADDAL` store to memory with release semantics.
- `LDADD` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs					0	0	0	0	0	0	Rn					Rt				
size																	opc														

32-bit, acquire (size == 10 && A == 1 && R == 0)

```
LDADDA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release (size == 10 && A == 1 && R == 1)

```
LDADDAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)

```
LDADD <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)

```
LDADDL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire (size == 11 && A == 1 && R == 0)

```
LDADDA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release (size == 11 && A == 1 && R == 1)

```
LDADDAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)

```
LDADD <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)

```
LDADDL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

result = data + value;
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDADDAB` and `LDADDALB` load from memory with acquire semantics.
- `LDADDLB` and `LDADDALB` store to memory with release semantics.
- `LDADDB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12 11	10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size													opc			

Acquire (A == 1 && R == 0)

```
LDADDAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDADDALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDADDB <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDADDLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, ldacctype];

result = data + value;
Mem[address, 1, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDADDAH` and `LDADDALH` load from memory with acquire semantics.
- `LDADDLH` and `LDADDALH` store to memory with release semantics.
- `LDADDH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	0	0	0	0	0	Rn					Rt				
size																	opc														

Acquire (A == 1 && R == 0)

```
LDADDAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDADDALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDADDH <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDADDLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, ldacctype];

result = data + value;
Mem[address, 2, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-Acquirepc or a Store-Release, created by having a Store-Release followed by a Load-Acquirepc instruction.
- The reading of a value written by a Store-Release by a Load-Acquirepc instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	1	0	1	(1)	(1)	(1)	(1)	(1)	1	1	0	0	0	0	Rn					Rt				
size											Rs																				

32-bit (size == 10)

```
LDAPR <Wt>, [<Xn|SP> {,#0}]
```

64-bit (size == 11)

```
LDAPR <Xt>, [<Xn|SP> {,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

AccType acctype = AccType_ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAPRB

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-Acquirepc or a Store-Release, created by having a Store-Release followed by a Load-Acquirepc instruction.
- The reading of a value written by a Store-Release by a Load-Acquirepc instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	1	0	1	(1)	(1)	(1)	(1)	(1)	1	1	0	0	0	0	Rn					Rt				
size											Rs																				

Integer

```
LDAPRB <Wt>, [<Xn|SP> {,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAPRH

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-Acquirepc or a Store-Release, created by having a Store-Release followed by a Load-Acquirepc instruction.
- The reading of a value written by a Store-Release by a Load-Acquirepc instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	0	1	(1)	(1)	(1)	(1)	(1)	1	1	0	0	0	0	Rn					Rt				
size											Rs																				

Integer

```
LDAPRH <Wt>, [<Xn|SP> {,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

32-bit (size == 10)

```
LDAR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
LDAR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, dbytes, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
LDARB <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
LDARH <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	sz	0	0	1	0	0	0	0	1	1	(1)	(1)	(1)	(1)	(1)	1	Rt2					Rn					Rt				
									L				Rs			o0															

32-bit (sz == 0)

```
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit (sz == 1)

```
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
```

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.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ORDERED];
    if BigEndian() then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_ORDERED, FALSE, FALSE));
    X[t] = Mem[address, 8, AccType_ORDERED];
    X[t2] = Mem[address+8, 8, AccType_ORDERED];
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

32-bit (size == 10)

```
LDAXR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
LDAXR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
LDAXRB <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
LDAXRH <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDCLRA` and `LDCLRAL` load from memory with acquire semantics.
- `LDCLRL` and `LDCLRAL` store to memory with release semantics.
- `LDCLR` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	0 0 1	0 0	Rn	Rt
size												opc				

32-bit, acquire (size == 10 && A == 1 && R == 0)

```
LDCLRA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release (size == 10 && A == 1 && R == 1)

```
LDCLRAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)

```
LDCLR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)

```
LDCLRL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire (size == 11 && A == 1 && R == 0)

```
LDCLRA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release (size == 11 && A == 1 && R == 1)

```
LDCLRAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)

```
LDCLR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)

```
LDCLRL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

result = data AND NOT(value);
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDCLRAB` and `LDCLRALB` load from memory with acquire semantics.
- `LDCLRLB` and `LDCLRALB` store to memory with release semantics.
- `LDCLRB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	0 0 1	0 0	Rn	Rt
size												opc				

Acquire (A == 1 && R == 0)

```
LDCLRAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDCLRALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDCLRB <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDCLRLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, ldacctype];

result = data AND NOT(value);
Mem[address, 1, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDCLRAH` and `LDCLRALH` load from memory with acquire semantics.
- `LDCLRLH` and `LDCLRALH` store to memory with release semantics.
- `LDCLRH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	0	0	1	0	0	Rn					Rt				
size																	opc														

Acquire (A == 1 && R == 0)

```
LDCLRAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDCLRALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDCLRH <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDCLRLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, ldacctype];

result = data AND NOT(value);
Mem[address, 2, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDEORA` and `LDEORAL` load from memory with acquire semantics.
- `LDEORL` and `LDEORAL` store to memory with release semantics.
- `LDEOR` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	0 1 0	0 0	Rn	Rt
size												opc				

32-bit, acquire (size == 10 && A == 1 && R == 0)

```
LDEORA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release (size == 10 && A == 1 && R == 1)

```
LDEORAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)

```
LDEOR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)

```
LDEORL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire (size == 11 && A == 1 && R == 0)

```
LDEORA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release (size == 11 && A == 1 && R == 1)

```
LDEORAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)

```
LDEOR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)

```
LDEORL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

result = data EOR value;
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDEORAB` and `LDEORALB` load from memory with acquire semantics.
- `LDEORLB` and `LDEORALB` store to memory with release semantics.
- `LDEORB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	0	1	0	0	0	Rn					Rt				
size																	opc														

Acquire (A == 1 && R == 0)

```
LDEORAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDEORALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDEORB <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, ldacctype];

result = data EOR value;
Mem[address, 1, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDEORAH` and `LDEORALH` load from memory with acquire semantics.
- `LDEORLH` and `LDEORALH` store to memory with release semantics.
- `LDEORH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	0	1	0	0	0	Rn					Rt				
size																	opc														

Acquire (A == 1 && R == 0)

```
LDEORAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDEORALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDEORH <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDEORLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, ldacctype];

result = data EOR value;
Mem[address, 2, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

32-bit (size == 10)

```
LDLAR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
LDLAR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, dbytes, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
LDLARB <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDLARH

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
LDLARH <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 2, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 28 27	26	25 24 23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
opc	1 0 1	1	0 0 0	1	imm7	Rt2	Rn	Rt
				L				

32-bit (opc == 00)

```
LDNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 01)

```
LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit (opc == 10)

```
LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDNP (SIMD&FP)*.

Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data1 = Mem[address, dbytes, AccType_VECSTREAM];
data2 = Mem[address+dbytes, dbytes, AccType_VECSTREAM];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
V[t] = data1;
V[t2] = data2;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see *Load/Store addressing modes*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair*.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
x	0	1	0	1	0	0	0	0	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDNP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data1 = Mem[address, dbytes, AccType_STREAM];
data2 = Mem[address+dbytes, dbytes, AccType_STREAM];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
X[t] = data1;
X[t2] = data2;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	0	1	1	imm7							Rt2					Rn						Rt			
									L																						

32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	1	1	1	imm7							Rt2					Rn						Rt			
									L																						

32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	1	0	1	imm7							Rt2					Rn						Rt			
									L																						

32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDP (SIMD&FP)*.

Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16. For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = Mem[address, dbytes, AccType_VEC];
data2 = Mem[address+dbytes, dbytes, AccType_VEC];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
V[t] = data1;
V[t2] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
x	0	1	0	1	0	0	0	1	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
x	0	1	0	1	0	0	1	1	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
x	0	1	0	1	0	0	1	0	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = Mem[address, dbytes, AccType_NORMAL];
data2 = Mem[address+dbytes, dbytes, AccType_NORMAL];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if signed then
    X[t] = SignExtend(data1, 64);
    X[t2] = SignExtend(data2, 64);
else
    X[t] = data1;
    X[t2] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	0	1	1	imm7							Rt2					Rn					Rt				
opc									L																						

Post-index

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	1	1	imm7							Rt2					Rn					Rt				
opc									L																						

Pre-index

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	0	1	imm7							Rt2					Rn					Rt				
opc									L																						

Signed offset

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDPSW*.

Assembler Symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
bits(64) offset = LSL(SignExtend(imm7, 64), 2);
```

Operation

```
bits(64) address;
bits(32) data1;
bits(32) data2;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = Mem[address, 4, AccType_NORMAL];
data2 = Mem[address+4, 4, AccType_NORMAL];
if rt_unknown then
    data1 = bits(32) UNKNOWN;
    data2 = bits(32) UNKNOWN;
X[t] = SignExtend(data1, 64);
X[t2] = SignExtend(data2, 64);
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;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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>], #<simm>
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>], #<simm>
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>], #<simm>
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>], #<simm>
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28 27	26	25 24	23 22	21	20 19 18 17 16 15 14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
size	1 1 1	1	0 0	x 1	0	imm9	1 1	Rn	Rt
				opc					

8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>, #<simm>]!
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, #<simm>]!
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>, #<simm>]!
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
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
size	1 1 1	1	0 1	x 1	imm12	Rn	Rt
				opc			

8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	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;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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>], #<simm>
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	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>, #<simm>]!
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
1	x	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size								opc				

32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDR (immediate)*.

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);

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 v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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'
        UnallocatedEncoding();

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Dt>	Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<label>	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

CheckFPAdvSIMDEnabled64();

data = Mem[address, size, AccType_VEC];
V[t] = data;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	x	0	1	1	0	0	0	imm19																			Rt				
opc																															

32-bit (opc == 00)

```
LDR <Wt>, <label>
```

64-bit (opc == 01)

```
LDR <Xt>, <label>
```

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<label>	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

case memop of
    when MemOp_LOAD
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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-bit (size == 00 && opc == 01 && option != 011)

```
LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

8-bit (size == 00 && opc == 01 && option == 011)

```
LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

128-bit (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 UnallocatedEncoding();
if option<1> == '0' then UnallocatedEncoding();     // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	For the 8-bit variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	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;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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>}}]
```

```
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding();    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRAA, LDRAB

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for `LDRAA`, and key B is used for `LDRAB`.

If the authentication passes, the PE behaves the same as for an `LDR` instruction. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	M	S	1	imm9									W	1	Rn					Rt				
size																															

Key A, offset (M == 0 && W == 0)

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]
```

Key A, pre-indexed (M == 0 && W == 1)

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!
```

Key B, offset (M == 1 && W == 0)

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]
```

Key B, pre-indexed (M == 1 && W == 1)

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!
```

```
if !HavePACExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) S10 = S:imm9;
bits(64) offset = LSL(SignExtend(S10, 64), 3);
```

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, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <simm>/8.

Operation

```
bits(64) address;
bits(64) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if use_key_a then
    address = AuthDA(address, X[31]);
else
    address = AuthDB(address, X[31]);

address = address + offset;
data = Mem[address, 8, AccType_NORMAL];
X[t] = data;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size								opc							

Post-index

```
LDRB <Wt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size								opc							

Pre-index

```
LDRB <Wt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size								opc				

Unsigned offset

```
LDRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

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);
```

Operation

```
bits(64) address;
bits(8) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

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;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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*.

31	30	29	28	27	26	25	24	23	22	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	1	Rm					option			S	1	0	Rn					Rt				
size								opc																							

Extended register (option != 011)

```
LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register (option == 011)

```
LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then UnallocatedEncoding();    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

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	SXTX

<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);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	1	0	imm9									0	1	Rn					Rt				
size								opc																							

Post-index

```
LDRH <Wt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	1	0	imm9									1	1	Rn					Rt				
size								opc																							

Pre-index

```
LDRH <Wt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
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																							

Unsigned offset

```
LDRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

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, 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);
```

Operation

```
bits(64) address;
bits(16) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

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;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																							

32-bit

```
LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UnallocatedEncoding();     // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	0	1	x	0	imm9	0	1	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>], #<simm>
```

64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	0	1	x	0	imm9	1	1	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	1	1	x	imm12	Rn	Rt
size								opc				

32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

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.
<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);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
```

Operation

```
bits(64) address;
bits(8) data;
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      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(8) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	Rm					option			S	1	0	Rn					Rt				
size								opc																							

32-bit with extended register offset (opc == 11 && option != 011)

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

32-bit with shifted register offset (opc == 11 && option == 011)

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

64-bit with extended register offset (opc == 10 && option != 011)

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

64-bit with shifted register offset (opc == 10 && option == 011)

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then UnallocatedEncoding();     // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

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	SXTX

<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);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
bits(64) address;
bits(8) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	x	0	imm9									0	1	Rn					Rt				
size								opc																							

32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>], #<simm>
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	x	0	imm9									1	1	Rn					Rt				
size								opc																							

32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	1	1	x	imm12												Rn					Rt				
size								opc																							

32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

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.
<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);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
```

Operation

```
bits(64) address;
bits(16) data;
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      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(16) UNKNOWN;
        else
            data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	x	1	Rm					option			S	1	0	Rn					Rt				
size								opc																							

32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UnallocatedEncoding();    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	0	1	1	1	0	0	0	1	0	0	imm9	0	1	Rn	Rt
size								opc							

Post-index

```
LDRSW <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	0	1	1	1	0	0	0	1	0	0	imm9	1	1	Rn	Rt
size								opc							

Pre-index

```
LDRSW <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
1	0	1	1	1	0	0	1	1	0	imm12	Rn	Rt
size								opc				

Unsigned offset

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);
```

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.
<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 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);
```

Operation

```
bits(64) address;
bits(32) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
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 v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	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	0	imm19																			Rt				
opc																															

Literal

```
LDRSW <Xt>, <label>
```

```
integer t = UInt(Rt);
bits(64) offset;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<label>	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(32) data;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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									

64-bit

```
LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UnallocatedEncoding();    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 2 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	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(32) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDSETA` and `LDSETAL` load from memory with acquire semantics.
- `LDSETL` and `LDSETAL` store to memory with release semantics.
- `LDSET` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	0 1 1	0 0	Rn	Rt
size												opc				

32-bit, acquire (size == 10 && A == 1 && R == 0)

```
LDSETA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release (size == 10 && A == 1 && R == 1)

```
LDSETAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)

```
LDSET <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)

```
LDSETL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire (size == 11 && A == 1 && R == 0)

```
LDSETA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release (size == 11 && A == 1 && R == 1)

```
LDSETAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)

```
LDSET <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)

```
LDSETL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

result = data OR value;
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDSETAB` and `LDSETALB` load from memory with acquire semantics.
- `LDSETLB` and `LDSETALB` store to memory with release semantics.
- `LDSETB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	0	1	1	0	0	Rn					Rt				
size																	opc														

Acquire (A == 1 && R == 0)

```
LDSETAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDSETALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSETB <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDSETLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, ldacctype];

result = data OR value;
Mem[address, 1, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDSETAH` and `LDSETALH` load from memory with acquire semantics.
- `LDSETLH` and `LDSETALH` store to memory with release semantics.
- `LDSETH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	0	1	1	0	0	Rn					Rt				
size																	opc														

Acquire (A == 1 && R == 0)

```
LDSETAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDSETALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSETH <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDSETLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, ldacctype];

result = data OR value;
Mem[address, 2, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDSMAXA` and `LDSMAXAL` load from memory with acquire semantics.
- `LDSMAXL` and `LDSMAXAL` store to memory with release semantics.
- `LDSMAX` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size												opc				

32-bit, acquire (size == 10 && A == 1 && R == 0)

```
LDSMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release (size == 10 && A == 1 && R == 1)

```
LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)

```
LDSMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)

```
LDSMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire (size == 11 && A == 1 && R == 0)

```
LDSMAXA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release (size == 11 && A == 1 && R == 1)

```
LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)

```
LDSMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)

```
LDSMAXL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

result = if SInt(data) > SInt(value) then data else value;
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDSMAXAB` and `LDSMAXALB` load from memory with acquire semantics.
- `LDSMAXLB` and `LDSMAXALB` store to memory with release semantics.
- `LDSMAXB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size												opc				

Acquire (A == 1 && R == 0)

```
LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSMAXB <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, ldacctype];

result = if SInt(data) > SInt(value) then data else value;
Mem[address, 1, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDSMAXAH` and `LDSMAXALH` load from memory with acquire semantics.
- `LDSMAXLH` and `LDSMAXALH` store to memory with release semantics.
- `LDSMAXH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size												opc				

Acquire (A == 1 && R == 0)

```
LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSMAXH <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, ldacctype];

result = if SInt(data) > SInt(value) then data else value;
Mem[address, 2, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDSMINA` and `LDSMINAL` load from memory with acquire semantics.
- `LDSMINL` and `LDSMINAL` store to memory with release semantics.
- `LDSMIN` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	1 0 1	0 0	Rn	Rt
size												opc				

32-bit, acquire (size == 10 && A == 1 && R == 0)

```
LDSMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release (size == 10 && A == 1 && R == 1)

```
LDSMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)

```
LDSMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)

```
LDSMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire (size == 11 && A == 1 && R == 0)

```
LDSMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release (size == 11 && A == 1 && R == 1)

```
LDSMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)

```
LDSMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)

```
LDSMINL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

result = if SInt(data) > SInt(value) then value else data;
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDSMINAB` and `LDSMINALB` load from memory with acquire semantics.
- `LDSMINLB` and `LDSMINALB` store to memory with release semantics.
- `LDSMINB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	1	0	1	0	0	Rn	Rt
size												opc							

Acquire (A == 1 && R == 0)

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, ldacctype];

result = if SInt(data) > SInt(value) then value else data;
Mem[address, 1, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDSMINAH` and `LDSMINALH` load from memory with acquire semantics.
- `LDSMINLH` and `LDSMINALH` store to memory with release semantics.
- `LDSMINH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					0	1	0	1	0	0	Rn					Rt				
size																	opc														

Acquire (A == 1 && R == 0)

```
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDSMINH <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, ldacctype];

result = if SInt(data) > SInt(value) then value else data;
Mem[address, 2, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for 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>{, #<simm>}]
```

64-bit (size == 11)

```
LDTR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
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_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    acctype = AccType_NORMAL;
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, acctype];
X[t] = ZeroExtend(data, regsize);
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for 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																							

Unscaled offset

```
LDTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

```
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_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    acctype = AccType_NORMAL;
```

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, acctype];
X[t] = ZeroExtend(data, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for 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					

Unscaled offset

```
LDTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

```
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_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    acctype = AccType_NORMAL;
```

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 2, acctype];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for 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>{, #<simm>}]
```

64-bit (opc == 10)

```
LDTRSB <Xt>, [<Xn|SP>{, #<simm>}]
```

```
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_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
```

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, 1, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for 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>{, #<simm>}]
```

64-bit (opc == 10)

```
LDTRSH <Xt>, [<Xn|SP>{, #<simm>}]
```

```
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_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
```

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, 2, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for 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					

Unscaled offset

```
LDTRSW <Xt>, [<Xn|SP>{, #<simm>}]
```

```
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_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    acctype = AccType_NORMAL;
```

Operation

```
bits(64) address;
bits(32) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 4, acctype];
X[t] = SignExtend(data, 64);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDUMAXA` and `LDUMAXAL` load from memory with acquire semantics.
- `LDUMAXL` and `LDUMAXAL` store to memory with release semantics.
- `LDUMAX` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	1	1	0	0	0	Rn	Rt
size												opc							

32-bit, acquire (size == 10 && A == 1 && R == 0)

```
LDUMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release (size == 10 && A == 1 && R == 1)

```
LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)

```
LDUMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)

```
LDUMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire (size == 11 && A == 1 && R == 0)

```
LDUMAXA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release (size == 11 && A == 1 && R == 1)

```
LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)

```
LDUMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)

```
LDUMAXL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

result = if UInt(data) > UInt(value) then data else value;
Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDUMAXAB` and `LDUMAXALB` load from memory with acquire semantics.
- `LDUMAXLB` and `LDUMAXALB` store to memory with release semantics.
- `LDUMAXB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					0	1	1	0	0	0	Rn					Rt				
size																	opc														

Acquire (A == 1 && R == 0)

```
LDUMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, ldacctype];

result = if UInt(data) > UInt(value) then data else value;
Mem[address, 1, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDUMAXAH` and `LDUMAXALH` load from memory with acquire semantics.
- `LDUMAXLH` and `LDUMAXALH` store to memory with release semantics.
- `LDUMAXH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	1 1 0	0 0	Rn	Rt
size												opc				

Acquire (A == 1 && R == 0)

```
LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDUMAXH <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, ldacctype];

result = if UInt(data) > UInt(value) then data else value;
Mem[address, 2, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDUMINA` and `LDUMINAL` load from memory with acquire semantics.
- `LDUMINL` and `LDUMINAL` store to memory with release semantics.
- `LDUMIN` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	1 1 1	0 0	Rn	Rt
size												opc				

32-bit, acquire (size == 10 && A == 1 && R == 0)

```
LDUMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release (size == 10 && A == 1 && R == 1)

```
LDUMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering (size == 10 && A == 0 && R == 0 && Rt != 11111)

```
LDUMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release (size == 10 && A == 0 && R == 1 && Rt != 11111)

```
LDUMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire (size == 11 && A == 1 && R == 0)

```
LDUMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release (size == 11 && A == 1 && R == 1)

```
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && A == 0 && R == 0 && Rt != 11111)

```
LDUMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release (size == 11 && A == 0 && R == 1 && Rt != 11111)

```
LDUMINL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, ldacctype];

result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDUMINAB` and `LDUMINALB` load from memory with acquire semantics.
- `LDUMINLB` and `LDUMINALB` store to memory with release semantics.
- `LDUMINB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
0	0	1	1	1	0	0	0	A	R	1	Rs	0	1 1 1	0 0	Rn	Rt
size												opc				

Acquire (A == 1 && R == 0)

```
LDUMINAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDUMINALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDUMINB <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDUMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, ldacctype];

result = if UInt(data) > UInt(value) then value else data;

Mem[address, 1, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDUMINAH` and `LDUMINALH` load from memory with acquire semantics.
- `LDUMINLH` and `LDUMINALH` store to memory with release semantics.
- `LDUMINH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	1 1 1	0 0	Rn	Rt
size												opc				

Acquire (A == 1 && R == 0)

```
LDUMINAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
LDUMINALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0 && Rt != 11111)

```
LDUMINH <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1 && Rt != 11111)

```
LDUMINLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, ldacctype];

result = if UInt(data) > UInt(value) then value else data;

Mem[address, 2, stacctype] = result;

X[t] = ZeroExtend(data, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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>{, #<simm>}]
```

16-bit (size == 01 && opc == 01)

```
LDUR <Ht>, [<Xn|SP>{, #<simm>}]
```

32-bit (size == 10 && opc == 01)

```
LDUR <St>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11 && opc == 01)

```
LDUR <Dt>, [<Xn|SP>{, #<simm>}]
```

128-bit (size == 00 && opc == 11)

```
LDUR <Qt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<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_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	1	0	imm9									0	0	Rn					Rt				
size								opc																							

Unscaled offset

```
LDURB <Wt>, [<Xn|SP>{, #<simm>}]
```

```
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);
```

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	1	0	imm9									0	0	Rn					Rt				
size								opc																							

Unscaled offset

```
LDURH <Wt>, [<Xn|SP>{, #<simm>}]
```

```
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);
```

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	1	x	0	imm9									0	0	Rn					Rt				
size								opc																							

32-bit (opc == 11)

```
LDURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = 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);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
```

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	x	0	imm9									0	0	Rn					Rt				
size								opc																							

32-bit (opc == 11)

```
LDURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = 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);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
```

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	0	1	1	1	0	0	0	1	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
LDURSW <Xt>, [<Xn|SP>{, #<simm>}]
```

```
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);
```

Operation

```
bits(64) address;
bits(32) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	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);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
```

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.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ATOMIC];
    if BigEndian() then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_ATOMIC, FALSE, FALSE));
    X[t] = Mem[address, 8, AccType_ATOMIC];
    X[t2] = Mem[address+8, 8, AccType_ATOMIC];
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

32-bit (size == 10)

```
LDXR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
LDXR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ATOMIC];
X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
LDXRB <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
LDXRH <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This is an alias of LSLV. This means:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0 0 1 0	0 0	Rn	Rd
													op2		

32-bit (sf == 0)

```
LSL <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSLV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
LSL <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSLV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of LSLV gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1	0	1	0	0	1	1	0	N	immr	!= x11111	Rn	Rd
	opc										imms		

32-bit (sf == 0 && N == 0 && imms != 011111)

```
LSL <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<shift> MOD 32), #(31-<shift>)
```

and is the preferred disassembly when `imms + 1 == immr`.

64-bit (sf == 1 && N == 1 && imms != 111111)

```
LSL <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)
```

and is the preferred disassembly when `imms + 1 == immr`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift>	For the 32-bit variant: is the shift amount, in the range 0 to 31. For the 64-bit variant: is the shift amount, in the range 0 to 63.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSLV

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register).

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0 0 1 0	0 0	Rn	Rd
													op2		

32-bit (sf == 0)

```
LSLV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
LSLV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of LSRV. This means:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0	0	1	Rn					Rd				
																				op2											

32-bit (sf == 0)

```
LSR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSRV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
LSR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSRV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of LSRV gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	0	0	1	1	0	N	immr						x	1	1	1	1	1	Rn					Rd				
	opc																	imms													

32-bit (sf == 0 && N == 0 && imms == 011111)

```
LSR <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1 && imms == 111111)

```
LSR <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift>	For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSRV

Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0	0	1	Rn					Rd				
																				op2											

32-bit (sf == 0)

```
LSRV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
LSRV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	1	0	1	1	0	0	0	Rm	o0	Ra	Rn	Rd

32-bit (sf == 0)

```
MADD <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit (sf == 1)

```
MADD <Xd>, <Xn>, <Xm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa>	Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
MUL	`Ra == '11111'`

Operation

```
bits(destsize) operand1 = X[n];
bits(destsize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d] = result<destsize-1:0>;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLA (by element)

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				0	0	0	0	H	0	Rn					Rd				
																	o2														

Vector

```
MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					1	0	0	1	0	1	Rn					Rd				
		U																													

Three registers of the same type

```
MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	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	0	0	H	0	Rn					Rd				
																	o2														

Vector

```
MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLS (vector)

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					1	0	0	1	0	1	Rn					Rd				
		U																													

Three registers of the same type

```
MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This is an alias of MSUB. This means:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	1	0	1	1	0	0	0	Rm	1	1	1	1	1	1	Rn	Rd
												o0			Ra				

32-bit (sf == 0)

```
MNEG <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MSUB <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
MNEG <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MSUB <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MSUB gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (to/from SP)

Move between register and stack pointer: `Rd = Rn`.

This is an alias of ADD (immediate). This means:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Rn					Rd				
op		S						shift							imm12																

32-bit (sf == 0)

```
MOV <Wd|WSP>, <Wn|WSP>
```

is equivalent to

```
ADD <Wd|WSP>, <Wn|WSP>, #0
```

and is the preferred disassembly when `(Rd == '11111' || Rn == '11111')`.

64-bit (sf == 1)

```
MOV <Xd|SP>, <Xn|SP>
```

is equivalent to

```
ADD <Xd|SP>, <Xn|SP>, #0
```

and is the preferred disassembly when `(Rd == '11111' || Rn == '11111')`.

Assembler Symbols

<Wd|WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn|WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Operation

The description of ADD (immediate) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (scalar)

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, 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.

This is an alias of DUP (element). This means:

- The encodings in this description are named to match the encodings of DUP (element).
- The description of DUP (element) 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	1	0	1	1	1	1	0	0	0	0	imm5	0	0 0 0 0	1	Rn	Rd

Scalar

```
MOV <V><d>, <Vn>.<T>[<index>]
```

is equivalent to

```
DUP <V><d>, <Vn>.<T>[<index>]
```

and is always the preferred disassembly.

Assembler Symbols

<V> Is the destination width specifier, encoded in “imm5”:

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the element width specifier, encoded in “imm5”:

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

Operation

The description of DUP (element) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (element)

Move vector element to another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the *CPACR_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 INS (element). This means:

- The encodings in this description are named to match the encodings of INS (element).
- The description of INS (element) 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	1	1	0	1	1	1	0	0	0	0	imm5	0	imm4	1	Rn	Rd

Advanced SIMD

```
MOV <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

is equivalent to

```
INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index1> Is the destination element index encoded in “imm5”:

imm5	<index1>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index2> Is the source element index encoded in “imm5:imm4”:

imm5	<index2>
x0000	RESERVED
xxxx1	imm4<3:0>
xxx10	imm4<3:1>
xx100	imm4<3:2>
x1000	imm4<3>

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

The description of INS (element) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (from general)

Move general-purpose register to a vector element. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (general). This means:

- The encodings in this description are named to match the encodings of INS (general).
- The description of INS (general) 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	1	0	0	1	1	1	0	0	0	0	imm5	0	0	0	1	1	1	Rn	Rd

Advanced SIMD

```
MOV <Vd>.<Ts>[<index>], <R><n>
```

is equivalent to

```
INS <Vd>.<Ts>[<index>], <R><n>
```

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

The description of INS (general) gives the operational pseudocode for this instruction.

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This is an alias of MOVN. This means:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	1	0	1	hw		imm16																Rd				
	opc																														

32-bit (sf == 0)

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVN <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)`.

64-bit (sf == 1)

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVN <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xffff0000 and 0x0000ffff For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVN gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	0	0	1	0	1	hw		imm16																Rd				
	opc																														

32-bit (sf == 0)

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVZ <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

64-bit (sf == 1)

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVZ <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw". For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw".
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (vector)

Move vector. This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of ORR (vector, register). This means:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) 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	0	1	0	1	Rm	0	0	0	1	1	1	Rn	Rd
								size											

Three registers of the same type

```
MOV <Vd>.<T>, <Vn>.<T>
```

is equivalent to

```
ORR <Vd>.<T>, <Vn>.<T>, <Vn>.<T>
```

and is the preferred disassembly when `Rm == Rn`.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This is an alias of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9	8	7	6	5	4 3 2 1 0
sf	0	1	1	0	0	1	0	0	N	immr	imms	1	1	1	1	1	Rd
	opc													Rn			

32-bit (sf == 0 && N == 0)

```
MOV <Wd|WSP>, #<imm>
```

is equivalent to

```
ORR <Wd|WSP>, WZR, #<imm>
```

and is the preferred disassembly when `! MoveWidePreferred(sf, N, imms, immr)`.

64-bit (sf == 1)

```
MOV <Xd|SP>, #<imm>
```

is equivalent to

```
ORR <Xd|SP>, XZR, #<imm>
```

and is the preferred disassembly when `! MoveWidePreferred(sf, N, imms, immr)`.

Assembler Symbols

<Wd|WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN. For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (register)

Move (register) copies the value in a source register to the destination register.

This is an alias of ORR (shifted register). This means:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	0	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	1	1	1	1	1	Rd				
opc								shift		N						imm6								Rn							

32-bit (sf == 0)

```
MOV <Wd>, <Wm>
```

is equivalent to

```
ORR <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
MOV <Xd>, <Xm>
```

is equivalent to

```
ORR <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wm>	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xm>	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (to general)

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of UMOV. This means:

- The encodings in this description are named to match the encodings of UMOV.
- The description of UMOV 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	0	0	0	0	x	x	x	0	0	0	0	1	1	1	1	Rn					Rd				
											imm5																				

32-bit (Q == 0 && imm5 == xx100)

```
MOV <Wd>, <Vn>.S[<index>]
```

is equivalent to

```
UMOV <Wd>, <Vn>.S[<index>]
```

and is always the preferred disassembly.

64-bit (Q == 1 && imm5 == x1000)

```
MOV <Xd>, <Vn>.D[<index>]
```

is equivalent to

```
UMOV <Xd>, <Vn>.D[<index>]
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<index>	For the 32-bit variant: is the element index encoded in "imm5<4:3>". For the 64-bit variant: is the element index encoded in "imm5<4>".

Operation

The description of UMOV gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOVI

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD&FP register. Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 14 13 12	11	10	9	8	7	6	5	4 3 2 1 0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode	0	1	d	e	f	g	h	Rd

8-bit (op == 0 && cmode == 1110)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #0}
```

16-bit shifted immediate (op == 0 && cmode == 10x0)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifted immediate (op == 0 && cmode == 0xx0)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifting ones (op == 0 && cmode == 110x)

```
MOVI <Vd>.<T>, #<imm8>, MSL #<amount>
```

64-bit scalar (Q == 0 && op == 1 && cmode == 1110)

```
MOVI <Dd>, #<imm>
```

64-bit vector (Q == 1 && op == 1 && cmode == 1110)

```
MOVI <Vd>.2D, #<imm>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<imm> Is a 64-bit immediate 'aaaaaaaabbbbbbbbcccccccc dddddddd eeeeeeeeffffffffgggggggghhhhhhhh', encoded in "a:b:c:d:e:f:g:h".

<T> For the 8-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.
For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.
For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

31	30	29	28	27	26	25	24	23	22	21	20 … 5	4 … 0
sf	1	1	1	0	0	1	0	1	hw		imm16	Rd
	opc											

32-bit (sf == 0)

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

```
bits(datasize) result;

result = X[d];
result<pos+15:pos> = imm16;
X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOVN

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (inverted wide immediate).

31	30	29	28	27	26	25	24	23	22	21	20 … 5	4 … 0
sf	0	0	1	0	0	1	0	1	hw		imm16	Rd
	opc											

32-bit (sf == 0)

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVN <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Alias Conditions

Alias	Of variant	Is preferred when
MOV (inverted wide immediate)	64-bit	`! (IsZero(imm16) && hw != '00')`
MOV (inverted wide immediate)	32-bit	`! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)`

Operation

```
bits(datasize) result;

result = Zeros();

result<pos+15:pos> = imm16;
result = NOT(result);
X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOVZ

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (wide immediate).

31	30	29	28	27	26	25	24	23	22	21	20 … 5	4 … 0
sf	1	0	1	0	0	1	0	1	hw		imm16	Rd
	opc											

32-bit (sf == 0)

```
MOVZ <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Alias Conditions

Alias	Is preferred when
MOV (wide immediate)	`! (IsZero(imm16) && hw != '00')`

Operation

```
bits(datasize) result;

result = Zeros();

result<pos+15:pos> = imm16;
X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MRS

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	1	1	o0	op1			CRn				CRm				op2			Rt				
										L																					

System

```
MRS <Xt>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)
```

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".
The System register names are defined in *'AArch64 System Registers' in the System Register XML*.

<op0> Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

```
X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see *Process state, PSTATE*.

The bits that can be written are D, A, I, F, and SP. This set of bits is expanded in extensions to the architecture as follows:

- ARMv8.1 adds the PAN bit.
- ARMv8.2 adds the UAO bit.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	op1			0	1	0	0	CRm				op2			1	1	1	1	1

System

```
MSR <pstatefield>, #<imm>
```

```
AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');

bits(4) operand = CRm;
PSTATEField field;
case op1:op2 of
    when '000 011'
        if !HaveUAOExt() then
            UnallocatedEncoding();
        field = PSTATEField_UAO;
    when '000 100'
        if !HavePANExt() then
            UnallocatedEncoding();
        field = PSTATEField_PAN;
    when '000 101' field = PSTATEField_SP;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    otherwise UnallocatedEncoding();

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if op1 == '011' && PSTATE.EL == EL0 && (IsInHost() || SCTLR_EL1.UMA == '0') then
    AArch64.SystemRegisterTrap(EL1, '00', op2, op1, '0100', '11111', CRm, '0');
```

Assembler Symbols

<pstatefield> Is a PSTATE field name, encoded in “op1:op2”:

op1	op2	<pstatefield>	Architectural Feature
000	00x	RESERVED	-
000	010	RESERVED	-
000	011	UAO	ARMv8.2-UAO
000	100	PAN	ARMv8.1-PAN
000	101	SPSel	-
000	11x	RESERVED	-
001	xxx	RESERVED	-
010	xxx	RESERVED	-
011	0xx	RESERVED	-
011	10x	RESERVED	-
011	110	DAIFSet	-
011	111	DAIFClr	-
1xx	xxx	RESERVED	-

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
case field of
    when PSTATEField_SP
        PSTATE.SP = operand<0>;
    when PSTATEField_DAIFSet
        PSTATE.D = PSTATE.D OR operand<3>;
        PSTATE.A = PSTATE.A OR operand<2>;
        PSTATE.I = PSTATE.I OR operand<1>;
        PSTATE.F = PSTATE.F OR operand<0>;
    when PSTATEField_DAIFClr
        PSTATE.D = PSTATE.D AND NOT(operand<3>);
        PSTATE.A = PSTATE.A AND NOT(operand<2>);
        PSTATE.I = PSTATE.I AND NOT(operand<1>);
        PSTATE.F = PSTATE.F AND NOT(operand<0>);
    when PSTATEField_PAN
        PSTATE.PAN = operand<0>;
    when PSTATEField_UAO
        PSTATE.UAO = operand<0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	1	o0	op1			CRn				CRm				op2			Rt				
										L																					

System

```
MSR (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>
```

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".
The System register names are defined in *'AArch64 System Registers' in the System Register XML*.

<op0> Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

```
AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	1	0	1	1	0	0	0	Rm	1	Ra	Rn	Rd
												o0			

32-bit (sf == 0)

```
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit (sf == 1)

```
MSUB <Xd>, <Xn>, <Xm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa>	Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
MNEG	`Ra == '11111'`

Operation

```
bits(destsize) operand1 = X[n];
bits(destsize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
X[d] = result<destsize-1:0>;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MUL (by element)

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size		L	M	Rm				1	0	0	0	H	0	Rn					Rd				

Vector

```
MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MUL (vector)

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					1	0	0	1	1	1	Rn					Rd				
		U																													

Three registers of the same type

```
MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MUL

Multiply: `Rd = Rn * Rm`.

This is an alias of MADD. This means:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	1	0	0	0	Rm					0	1	1	1	1	1	Rn					Rd				
																o0			Ra												

32-bit (sf == 0)

```
MUL <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MADD <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
MUL <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MADD <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MADD gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVN

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of NOT. This means:

- The encodings in this description are named to match the encodings of NOT.
- The description of NOT 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	0	0	0	1	0	0	0	0	0	0	1	0	1	1	0	Rn					Rd				

Vector

```
MVN <Vd>.<T>, <Vn>.<T>
```

is equivalent to

```
NOT <Vd>.<T>, <Vn>.<T>
```

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of NOT gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVN

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This is an alias of ORN (shifted register). This means:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	0	1	0	1	0	shift		1	Rm					imm6						1	1	1	1	1	Rd				
	opc									N														Rn							

32-bit (sf == 0)

```
MVN <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
MVN <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVNI

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 14 13 12	11	10	9	8	7	6	5	4 3 2 1 0
0	Q	1	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode	0	1	d	e	f	g	h	Rd
		op																						

16-bit shifted immediate (cmode == 10x0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifted immediate (cmode == 0xx0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifting ones (cmode == 110x)

```
MVNI <Vd>.<T>, #<imm8>, MSL #<amount>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UnallocatedEncoding();
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.
For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.
For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NEG (vector)

Negate (vector). This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	0	0	0	0	0	1	0	1	1	1	0	Rn					Rd				
		U																													

Scalar

```
NEG <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	1	0	1	1	1	0	Rn					Rd				
		U																													

Vector

```
NEG <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This is an alias of SUB (shifted register). This means:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	0	1	0	1	1	shift		0	Rm					imm6						1	1	1	1	1	Rd				
	op	S																						Rn							

32-bit (sf == 0)

```
NEG <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUB <Wd>, WZR, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
NEG <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUB <Xd>, XZR, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NEGS

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	0	1	0	1	1	shift		0	Rm					imm6						1	1	1	1	1	Rd				
	op	S																						Rn							

32-bit (sf == 0)

```
NEGS <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Wd>, WZR, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
NEGS <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Xd>, XZR, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This is an alias of SBC. This means:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	1	1	1	1	1	Rd				
	op	S																						Rn							

32-bit (sf == 0)

```
NGC <Wd>, <Wm>
```

is equivalent to

```
SBC <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
NGC <Xd>, <Xm>
```

is equivalent to

```
SBC <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wm>	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xm>	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SBCS. This means:

- The encodings in this description are named to match the encodings of SBCS.
- The description of SBCS gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	1	1	1	1	1	Rd				
	op	S																						Rn							

32-bit (sf == 0)

```
NGCS <Wd>, <Wm>
```

is equivalent to

```
SBCS <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
NGCS <Xd>, <Xm>
```

is equivalent to

```
SBCS <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wm>	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xm>	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBCS gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

The timing effects of including a `NOP` instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, `NOP` instructions are not suitable for timing loops.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1
																					CRm				op2						

System

```
NOP
```

```
// Empty.
```

Operation

```
// do nothing
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NOT

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MVN.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	1	0	1	1	0	Rn					Rd				

Vector

```
NOT <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = NOT(element);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	Rm					0	0	0	1	1	1	Rn					Rd				
								size																							

Three registers of the same type

```
ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 OR operand2;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN.

31	30 29	28 27 26 25 24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0 1	0 1 0 1 0	shift	1	Rm	imm6	Rn	Rd
	opc			N				

32-bit (sf == 0)

```
ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
ORN <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Alias Conditions

Alias	Is preferred when
MVN	`Rn == '11111'`

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);

result = operand1 OR operand2;
X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	x	x	x	1	0	1	d	e	f	g	h	Rd				
		op														cmode															

16-bit (cmode == 10x1)

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit (cmode == 0xx1)

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x10' operation = ImmediateOp_ORR;
    when '110x0' operation = ImmediateOp_MOVI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR (vector, register)

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (vector).

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	1	0	1	Rm	0	0	0	1	1	1	Rn	Rd
								size											

Three registers of the same type

```
ORR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
MOV (vector)	`Rm == Rn`

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = operand1 OR operand2;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias MOV (bitmask immediate).

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	1	1	0	0	1	0	0	N	immr	imms	Rn	Rd
	opc												

32-bit (sf == 0 && N == 0)

```
ORR <Wd|WSP>, <Wn>, #<imm>
```

64-bit (sf == 1)

```
ORR <Xd|SP>, <Xn>, #<imm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Assembler Symbols

<Wd|WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Alias Conditions

Alias	Is preferred when
MOV (bitmask immediate)	`Rn == '11111' && ! MoveWidePreferred(sf, N, imms, immr)`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];

result = operand1 OR imm;
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR (shifted register)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MOV (register).

31	30 29	28 27 26 25	24	23 22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0 1	0 1 0 1	0	shift	0	Rm	imm6	Rn	Rd
	opc				N				

32-bit (sf == 0)

```
ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Alias Conditions

Alias	Is preferred when
MOV (register)	`shift == '00' && imm6 == '000000' && Rn == '11111'`

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 OR operand2;
X[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PACDA, PACDZA

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDA.
- The value zero, for PACDZA.

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	0	Rn					Rd				

PACDA (Z == 0)

```
PACDA <Xd>, <Xn|SP>
```

PACDZA (Z == 1 && Rn == 11111)

```
PACDZA <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UnallocatedEncoding();

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then UnallocatedEncoding();
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d] = AddPACDA(X[d], SP[]);
else
    X[d] = AddPACDA(X[d], X[n]);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PACDB, PACDZB

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for `PACDB`.
- The value zero, for `PACDZB`.

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	1	Rn					Rd				

PACDB (Z == 0)

```
PACDB <Xd>, <Xn|SP>
```

PACDZB (Z == 1 && Rn == 11111)

```
PACDZB <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UnallocatedEncoding();

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then UnallocatedEncoding();
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d] = AddPACDB(X[d], SP[]);
else
    X[d] = AddPACDB(X[d], X[n]);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PACGA

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of the destination register.

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	1	0	0	Rn					Rd				

Integer

```
PACGA <Xd>, <Xn>, <Xm|SP>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if !HavePACExt() then
    UnallocatedEncoding();

if m == 31 then source_is_sp = TRUE;
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm|SP>	Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

Operation

```
if source_is_sp then
    X[d] = AddPACGA(X[n], SP[]);
else
    X[d] = AddPACGA(X[n], X[m]);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by <Xd> for `PACIA` and `PACIZA`.
- In X17, for `PACIA1716`.
- In X30, for `PACIASP` and `PACIAZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for `PACIA`.
- The value zero, for `PACIZA` and `PACIAZ`.
- In X16, for `PACIA1716`.
- In SP, for `PACIASP`.

It has encodings from 2 classes: Integer and System

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	0	0	Rn					Rd				

PACIA (Z == 0)

```
PACIA <Xd>, <Xn|SP>
```

PACIZA (Z == 1 && Rn == 11111)

```
PACIZA <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UnallocatedEncoding();

if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE;
else // PACIZA
    if n != 31 then UnallocatedEncoding();
```

System

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	0	x	1	1	1	1	1
																					CRm				op2						

PACIA1716 (CRm == 0001 && op2 == 000)

```
PACIA1716
```

PACIASP (CRm == 0011 && op2 == 001)

```
PACIASP
```

PACIAZ (CRm == 0011 && op2 == 000)

```
PACIAZ
```

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 000'    // PACIAZ
        d = 30;
        n = 31;
    when '0011 001'    // PACIASP
        d = 30;
        source_is_sp = TRUE;
    when '0001 000'    // PACIA1716
        d = 17;
        n = 16;
    when '0001 010' SEE "PACIB";
    when '0001 100' SEE "AUTIA";
    when '0001 110' SEE "AUTIB";
    when '0011 01x' SEE "PACIB";
    when '0011 10x' SEE "AUTIA";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AddPACIA(X[d], SP[]);
    else
        X[d] = AddPACIA(X[d], X[n]);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by <Xd> for `PACIB` and `PACIZB`.
- In X17, for `PACIB1716`.
- In X30, for `PACIBSP` and `PACIBZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for `PACIB`.
- The value zero, for `PACIZB` and `PACIBZ`.
- In X16, for `PACIB1716`.
- In SP, for `PACIBSP`.

It has encodings from 2 classes: Integer and System

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	0	1	Rn					Rd				

PACIB (Z == 0)

```
PACIB <Xd>, <Xn|SP>
```

PACIZB (Z == 1 && Rn == 11111)

```
PACIZB <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UnallocatedEncoding();

if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then UnallocatedEncoding();
```

System

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	1	x	1	1	1	1	1
																					CRm				op2						

PACIB1716 (CRm == 0001 && op2 == 010)

```
PACIB1716
```

PACIBSP (CRm == 0011 && op2 == 011)

```
PACIBSP
```

PACIBZ (CRm == 0011 && op2 == 010)

```
PACIBZ
```

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 010'    // PACIBZ
        d = 30;
        n = 31;
    when '0011 011'    // PACIBSP
        d = 30;
        source_is_sp = TRUE;
    when '0001 010'    // PACIB1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 100' SEE "AUTIA";
    when '0001 110' SEE "AUTIB";
    when '0011 00x' SEE "PACIA";
    when '0011 10x' SEE "AUTIA";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d] = AddPACIB(X[d], SP[]);
    else
        X[d] = AddPACIB(X[d], X[n]);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PMUL

Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					1	0	0	1	1	1	Rn					Rd				
		U																													

Three registers of the same type

```
PMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PMULL, PMULL2

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}*.

The `PMULL` instruction extracts each source vector from the lower half of each source register, while the `PMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					1	1	1	0	0	0	Rn					Rd				

Three registers, not all the same type

```
PMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '01' || size == '10' then ReservedValue();
if size == '11' && !HaveCryptoExt() then UnallocatedEncoding();
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;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	imm12												Rn					Rt				
size								opc																							

Unsigned offset

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

```
bits(64) offset = LSL(ZeroExtend(imm12, 64), 3);
```

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);
```

Operation

```
bits(64) address;

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	imm19																			Rt				
opc																															

Literal

```
PRFM (<prfop>|#<imm5>), <label>
```

```
integer t = UInt(Rt);
bits(64) offset;

offset = SignExtend(imm19:'00', 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>.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;

Prefetch(address, t<4:0>);
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13	12	11	10	9 8 7 6 5	4 3 2 1 0
1	1	1	1	1	0	0	0	1	0	1	Rm	option	S	1	0	Rn	Rt
size								opc									

Integer

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UnallocatedEncoding();     // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 3 else 0;
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

PLD
Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI
Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST
Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1
Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2
Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3
Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP
Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM
Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFM (unscaled offset)

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an `PRFUM` instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	1	1	1	1	0	0	0	1	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}]
```

```
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);
```

Operation

```
bits(64) address;

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PSB CSYNC

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following `DSB` instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a `NOP`.

System

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	1	1	1	1	1	1
																					CRm				op2						

System

```
PSB CSYNC
```

```
if !HaveStatisticalProfiling() then EndOfInstruction();
```

Operation

```
ProfilingSynchronizationBarrier();
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RADDHN, RADDHN2

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see *ADDHN*.

The RADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	1	0	0	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
RADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RAX1

Rotate and Exclusive OR rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
1	1	0	0	1	1	1	0	0	1	1	Rm	1	0	0	0	1	1	Rn	Rd

Advanced SIMD

```
RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D
```

```
if !HaveCryptoExt2() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
V[d] = Vn EOR (ROL(Vm<127:64>, 1):ROL(Vm<63:0>, 1));
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RBIT (vector)

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	0	0	0	0	0	0	1	0	1	1	0	Rn					Rd				

Vector

```
RBIT <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
bits(esize) rev;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    for i = 0 to esize-1
        rev<esize-1-i> = element<i>;
    Elem[result, e, esize] = rev;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RBIT

Reverse Bits reverses the bit order in a register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	Rn					Rd				

32-bit (sf == 0)

```
RBIT <Wd>, <Wn>
```

64-bit (sf == 1)

```
RBIT <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

for i = 0 to datasize-1
    result<datasize-1-i> = operand<i>;

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	0	1	0	1	1	1	1	1	0	0	0	0	0	0	Rn					0	0	0	0	0
							Z		op											A	M								Rm		

Integer

```
RET {<Xn>}
```

```
integer n = UInt(Rn);
```

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

Operation

```
bits(64) target = X[n];
BranchTo(target, BranchType_RET);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RETAA, RETAB

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

Key A is used for `RETAA`, and key B is used for `RETAB`.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated. The authenticated address is not written back to LR.

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	0	1	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1
							Z		op											A				Rn					Rm		

RETAA (M == 0)

```
RETAA
```

RETAB (M == 1)

```
RETAB
```

```
boolean use_key_a = (M == '0');

if !HavePACExt() then
    UnallocatedEncoding();
```

Operation

```
bits(64) target = X[30];
bits(64) modifier = SP[];

if use_key_a then
    target = AuthIA(target, modifier);
else
    target = AuthIB(target, modifier);

BranchTo(target, BranchType_RET);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	x	Rn					Rd				
																				opc											

32-bit (sf == 0 && opc == 10)

```
REV <Wd>, <Wn>
```

64-bit (sf == 1 && opc == 11)

```
REV <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV16 (vector)

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	0	0	0	1	1	0	Rn					Rd				
		U																	o0												

Vector

```
REV16 <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	Rn					Rd				
																				opc											

32-bit (sf == 0)

```
REV16 <Wd>, <Wn>
```

64-bit (sf == 1)

```
REV16 <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV32 (vector)

Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	0	0	0	0	1	0	Rn					Rd				
		U																	o0												

Vector

```
REV32 <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV32

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	Rn					Rd				
sf																				opc											

64-bit

```
REV32 <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV64

Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	0	0	0	0	1	0	Rn					Rd				
		U																	o0												

Vector

```
REV64 <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UnallocatedEncoding();

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV64

Reverse Bytes reverses the byte order in a 64-bit general-purpose register.

When assembling for ARMv8.2, an assembler must support this pseudo-instruction. It is OPTIONAL whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than ARMv8.2.

This is a pseudo-instruction of REV. This means:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	Rn					Rd				
sf																				opc											

64-bit

```
REV64 <Xd>, <Xn>
```

is equivalent to

```
REV <Xd>, <Xn>
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of REV gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of EXTR. This means:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	0	0	1	1	1	N	0	Rm	imms	Rn	Rd

32-bit (sf == 0 && N == 0 && imms == 0xxxxx)

```
ROR <Wd>, <Ws>, #<shift>
```

is equivalent to

```
EXTR <Wd>, <Ws>, <Ws>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

64-bit (sf == 1 && N == 1)

```
ROR <Xd>, <Xs>, #<shift>
```

is equivalent to

```
EXTR <Xd>, <Xs>, <Xs>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Ws>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xs>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<shift>	For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

Operation

The description of EXTR gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of RORV. This means:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0	1	1	Rn					Rd				
																				op2											

32-bit (sf == 0)

```
ROR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
RORV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
ROR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
RORV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RORV

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0	1	1	Rn					Rd				
																				op2											

32-bit (sf == 0)

```
RORV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
RORV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSHRN, RSHRN2

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see *SHRN*.

The `RSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `RSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	0	0	1	1	Rn					Rd				
									immh											op										

Vector

```
RSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSUBHN, RSUBHN2

Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see *SUBHN*.

The `RSUBHN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `RSUBHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	1	1	0	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
RSUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABA

Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	1	1	1	1	Rn					Rd				
		U																		ac											

Three registers of the same type

```
SABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABAL, SABAL2

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `SABAL` instruction extracts each source vector from the lower half of each source register, while the `SABAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	0	1	0	0	Rn					Rd				
		U																op													

Three registers, not all the same type

```
SABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABD

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	1	1	0	1	Rn					Rd				
		U																		ac											

Three registers of the same type

```
SABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `SABDL` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SABDL2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	1	1	0	0	Rn					Rd				
		U																op													

Three registers, not all the same type

```
SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADALP

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	0	1	1	0	1	0	Rn					Rd				
		U															op														

Vector

```
SADALP <Vd>.<Ta>, <Vn>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDL, SADDL2

Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SADDL` instruction extracts each source vector from the lower half of each source register, while the `SADDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	0	0	0	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
SADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDLP

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	0	0	1	0	1	0	Rn					Rd				
		U															op														

Vector

```
SADDLP <Vd>.<Ta>, <Vn>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDLV

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	1	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
		U																													

Advanced SIMD

```
SADDLV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	H
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d] = sum<2*esize-1:0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDW, SADDW2

Signed Add Wide. This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The `SADDW` instruction extracts the second source vector from the lower half of the second source register, while the `SADDW2` instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	0	0	1	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
SADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
	op	S																													

32-bit (sf == 0)

```
SBC <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
SBC <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
NGC	`Rn == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

operand2 = NOT(operand2);

(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
	op	S																													

32-bit (sf == 0)

```
SBCS <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
SBCS <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
NGCS	`Rn == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;

operand2 = NOT(operand2);

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBFIZ

Signed Bitfield Insert in Zero zeroes the destination register and copies any number of contiguous bits from a source register into any position in the destination register, sign-extending the most significant bit of the transferred value.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	0	0	1	1	0	N	immr	imms	Rn	Rd
	opc												

32-bit (sf == 0 && N == 0)

```
SBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit (sf == 1 && N == 1)

```
SBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBFM

Signed Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, shifting in copies of the sign bit in the upper bits and zeros in the lower bits.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW.

31	30 29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0 0	1	0	0	1	1	0	N	immr	imms	Rn	Rd
	opc											

32-bit (sf == 0 && N == 0)

```
SBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit (sf == 1 && N == 1)

```
SBFM <Xd>, <Xn>, #<immr>, #<imms>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
<imms>	For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Of variant	Is preferred when
ASR (immediate)	32-bit	`imms == '011111'`
ASR (immediate)	64-bit	`imms == '111111'`
SBFIZ		`UInt(imms) < UInt(immr)`
SBFX		`BFXPreferred(sf, opc<1>, imms, immr)`
SXTB		`immr == '000000' && imms == '000111'`
SXTH		`immr == '000000' && imms == '001111'`

Alias	Of variant	Is preferred when
SXTW		`immr == '000000' && imms == '011111'`

Operation

```
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, R) AND wmask;

// determine extension bits (sign, zero or dest register)
bits(datasize) top = Replicate(src<S>);

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBFX

Signed Bitfield Extract extracts any number of adjacent bits at any position from a register, sign-extends them to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	0	0	1	0	0	1	1	0	N	immr	imms	Rn	Rd
	opc												

32-bit (sf == 0 && N == 0)

```
SBFX <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

64-bit (sf == 1 && N == 1)

```
SBFX <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SCVTF (vector, fixed-point)

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn					Rd				
		U							immh																						

Scalar

```
SCVTF <V><d>, <V><n>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn					Rd				
		U							immh																						

Vector

```
SCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<V>	Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SCVTF (vector, integer)

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U																													

Scalar half precision

```
SCVTF <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U																													

Scalar single-precision and double-precision

```
SCVTF <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U																													

Vector half precision

```
SCVTF <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U																													

Vector single-precision and double-precision

```
SCVTF <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
FPRounding rounding = FPRoundingMode(FPCR);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type		0	0	0	0	1	0	scale						Rn					Rd				
											rmode		opcode																		

32-bit to half-precision (sf == 0 && type == 11)
(ARMv8.2)

```
SCVTF <Hd>, <Wn>, #<fbits>
```

32-bit to single-precision (sf == 0 && type == 00)

```
SCVTF <Sd>, <Wn>, #<fbits>
```

32-bit to double-precision (sf == 0 && type == 01)

```
SCVTF <Dd>, <Wn>, #<fbits>
```

64-bit to half-precision (sf == 1 && type == 11)
(ARMv8.2)

```
SCVTF <Hd>, <Xn>, #<fbits>
```

64-bit to single-precision (sf == 1 && type == 00)

```
SCVTF <Sd>, <Xn>, #<fbits>
```

64-bit to double-precision (sf == 1 && type == 01)

```
SCVTF <Dd>, <Xn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale". For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = FixedToFP(intval, fracbits, FALSE, FPCR, rounding);
V[d] = fltval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	type		1	0	0	0	1	0	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

32-bit to half-precision (sf == 0 && type == 11)
(ARMv8.2)

```
SCVTF <Hd>, <Wn>
```

32-bit to single-precision (sf == 0 && type == 00)

```
SCVTF <Sd>, <Wn>
```

32-bit to double-precision (sf == 0 && type == 01)

```
SCVTF <Dd>, <Wn>
```

64-bit to half-precision (sf == 1 && type == 11)
(ARMv8.2)

```
SCVTF <Hd>, <Xn>
```

64-bit to single-precision (sf == 1 && type == 00)

```
SCVTF <Sd>, <Xn>
```

64-bit to double-precision (sf == 1 && type == 01)

```
SCVTF <Dd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = FixedToFP(intval, 0, FALSE, FPCR, rounding);
V[d] = fltval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	0	0	1	1	Rn					Rd				
																					o1										

32-bit (sf == 0)

```
SDIV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
SDIV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(Int(operand2, FALSE)));

X[d] = result<datasize-1:0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SDOT (by element)

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

From ARMv8.2, this is an OPTIONAL instruction.

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size		L	M	Rm				1	1	1	0	H	0	Rn					Rd				
		U																													

Vector

```
SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then ReservedValue();
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the element index, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SDOT (vector)

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

From ARMv8.2, this is an OPTIONAL instruction.

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Three registers of the same type

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		0	Rm					1	0	0	1	0	1	Rn					Rd				
		U																													

Three registers of the same type

```
SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then ReservedValue();
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	1	1	1
																					CRm				op2						

System

```
SEV
```

```
// Empty.
```

Operation

```
SendEvent();
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a `WFE` instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	1	1	1	1	1
																					CRm				op2						

System

```
SEVL
```

```
// Empty.
```

Operation

```
SendEventLocal();
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1C

SHA1 hash update (choose).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				

Advanced SIMD

```
SHA1C <Qd>, <Sn>, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckCryptoEnabled64();

bits(128) X = V[d];
bits(32) Y = V[n];     // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
    t = SHAchoose(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
V[d] = X;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1H

SHA1 fixed rotate.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	Rn					Rd				

Advanced SIMD

```
SHA1H <Sd>, <Sn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveCryptoExt() then UnallocatedEncoding();
```

Assembler Symbols

<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckCryptoEnabled64();

bits(32) operand = V[n];     // read element [0] only,  [1-3] zeroed
V[d] = ROL(operand, 30);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1M

SHA1 hash update (majority).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	0	1	0	0	0	Rn					Rd				

Advanced SIMD

```
SHA1M <Qd>, <Sn>, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckCryptoEnabled64();

bits(128) X = V[d];
bits(32) Y = V[n];     // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
    t = SHAmajority(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
V[d] = X;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1P

SHA1 hash update (parity).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	0	0	1	0	0	Rn					Rd				

Advanced SIMD

```
SHA1P <Qd>, <Sn>, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckCryptoEnabled64();

bits(128) X = V[d];
bits(32) Y = V[n];     // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
    t = SHAparity(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
V[d] = X;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1SU0

SHA1 schedule update 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	0	1	1	0	0	Rn					Rd				

Advanced SIMD

```
SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;

result = operand2<63:0>:operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1SU1

SHA1 schedule update 1.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	Rn					Rd				

Advanced SIMD

```
SHA1SU1 <Vd>.4S, <Vn>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveCryptoExt() then UnallocatedEncoding();
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand1 EOR LSR(operand2, 32);
result<31:0> = ROL(T<31:0>, 1);
result<63:32> = ROL(T<63:32>, 1);
result<95:64> = ROL(T<95:64>, 1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256H2

SHA256 hash update (part 2).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	1	0	1	0	0	Rn					Rd				
																			P												

Advanced SIMD

```
SHA256H2 <Qd>, <Qn>, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckCryptoEnabled64();

bits(128) result;
result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256H

SHA256 hash update (part 1).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	1	0	0	0	0	Rn					Rd				
																			P												

Advanced SIMD

```
SHA256H <Qd>, <Qn>, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckCryptoEnabled64();

bits(128) result;
result = SHA256hash(V[d], V[n], V[m], TRUE);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256SU0

SHA256 schedule update 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	Rn					Rd				

Advanced SIMD

```
SHA256SU0 <Vd>.4S, <Vn>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveCryptoExt() then UnallocatedEncoding();
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand2<31:0>:operand1<127:32>;
bits(32) elt;

for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256SU1

SHA256 schedule update 1.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	1	1	0	0	0	Rn					Rd				

Advanced SIMD

```
SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveCryptoExt() then UnallocatedEncoding();
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckCryptoEnabled64();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
bits(128) T0 = operand3<31:0>:operand2<127:32>;
bits(64) T1;
bits(32) elt;

T1 = operand3<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e-2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA512H2

SHA512 Hash update part 2 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm					1	0	0	0	0	1	Rn					Rd				

Advanced SIMD

```
SHA512H2 <Qd>, <Qn>, <Vm>.2D
```

```
if !HaveCryptoExt2() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
bits(128) Vtmp;
bits(64) NSigma0;
bits(64) tmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

NSigma0 = ROR(Y<63:0>, 28) EOR ROR(Y<63:0>, 34) EOR ROR(Y<63:0>, 39);
Vtmp<127:64> = (X<63:0> AND Y<127:64>) EOR (X<63:0> AND Y<63:0>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 + W<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR ROR(Vtmp<127:64>, 39);
Vtmp<63:0> = (Vtmp<127:64> AND Y<63:0>) EOR (Vtmp<127:64> AND Y<127:64>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + NSigma0 + W<63:0>);

V[d] = Vtmp;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA512H

SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register. This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13 12	11 10	9 8 7 6 5	4 3 2 1 0
1	1	0	0	1	1	1	0	0	1	1	Rm	1	0	0 0	0 0	Rn	Rd

Advanced SIMD

```
SHA512H <Qd>, <Qn>, <Vm>.2D
```

```
if !HaveCryptoExt2() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
bits(128) Vtmp;
bits(64) MSigma1;
bits(64) tmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

MSigma1 = ROR(Y<127:64>, 14) EOR ROR(Y<127:64>, 18) EOR ROR(Y<127:64>, 41);
Vtmp<127:64> = (Y<127:64> AND X<63:0>) EOR (NOT(Y<127:64>) AND X<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + W<127:64>);
tmp = Vtmp<127:64> + Y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR ROR(tmp, 41);
Vtmp<63:0> = (tmp AND Y<127:64>) EOR (NOT(tmp) AND X<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + W<63:0>);
V[d] = Vtmp;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA512SU0

SHA512 Schedule Update 0 takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	Rn					Rd				

Advanced SIMD

```
SHA512SU0 <Vd>.2D, <Vn>.2D
```

```
if !HaveCryptoExt2() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
bits(64) sig0;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) W = V[d];
sig0 = ROR(W<127:64>, 1) EOR ROR(W<127:64>, 8) EOR '0000000':W<127:71>;
Vtmp<63:0> = W<63:0> + sig0;
sig0 = ROR(X<63:0>, 1) EOR ROR(X<63:0>, 8) EOR '0000000':X<63:7>;
Vtmp<127:64> = W<127:64> + sig0;
V[d] = Vtmp;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA512SU1

SHA512 Schedule Update 1 takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
1	1	0	0	1	1	1	0	0	1	1	Rm	1	0	0	0	1	0	Rn	Rd

Advanced SIMD

```
SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D
```

```
if !HaveCryptoExt2() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
bits(64) sig1;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

sig1 = ROR(X<127:64>, 19) EOR ROR(X<127:64>, 61) EOR '000000':X<127:70>;
Vtmp<127:64> = W<127:64> + sig1 + Y<127:64>;
sig1 = ROR(X<63:0>, 19) EOR ROR(X<63:0>, 61) EOR '000000':X<63:6>;
Vtmp<63:0> = W<63:0> + sig1 + Y<63:0>;
V[d] = Vtmp;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHADD

Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see *SRHADD*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	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	1	Rn					Rd				
		U																													

Three registers of the same type

```
SHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHL

Shift Left (immediate). This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	1	0	1	0	1	Rn					Rd				
									immh																						

Scalar

```
SHL <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	1	0	1	0	1	Rn					Rd				
									immh																						

Vector

```
SHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = LSL(Elem[operand, e, esize], shift);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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				

Vector

```
SHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = esize;
boolean unsigned = FALSE;     // Or TRUE without change of functionality
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<shift> Is the left shift amount, which must be equal to the source element width in bits, encoded in “size”:

size	<shift>
00	8
01	16
10	32
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHRN, SHRN2

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see *RSHRN*.

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	0	0	0	1	Rn					Rd				
									immh											op										

Vector

```
SHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHSUB

Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	0	1	0	0	1	Rn					Rd				
		U																													

Three registers of the same type

```
SHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

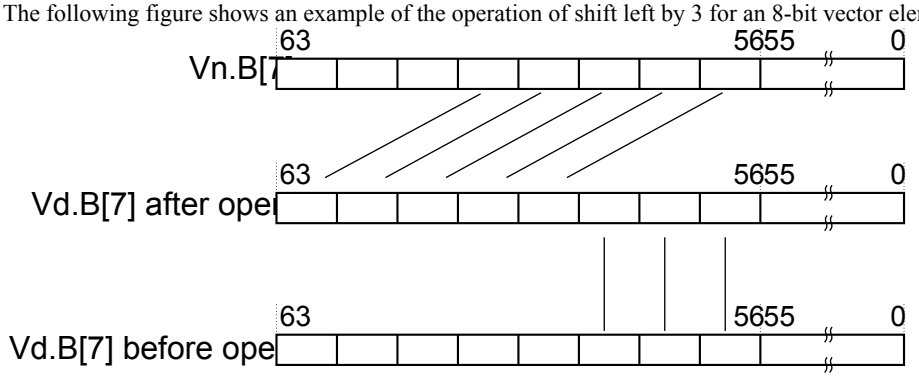
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SLI

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost. The following figure shows an example of the operation of shift left by 3 for an 8-bit vector element.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	1	0	1	0	1	Rn					Rd				
									immh																						

Scalar

```
SLI <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	0	1	0	1	Rn					Rd				
									immh																						

Vector

```
SLI <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
bits(esize) mask = LSL(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSL(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM3PARTW1

SM3PARTW1 takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm					1	1	0	0	0	0	Rn					Rd				

Advanced SIMD

```
SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

```
if !HaveChCryptoExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;

result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>, 15):ROL(Vm<95:64>, 15):ROL(Vm<63:32>, 15));

for i = 0 to 3
    if i == 3 then
        result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
    result<(32*i)+31:(32*i)> = result<(32*i)+31:(32*i)> EOR ROL(result<(32*i)+31:(32*i)>, 15) EOR ROL(r
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm					1	1	0	0	0	1	Rn					Rd				

Advanced SIMD

```
SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

```
if !HaveChCryptoExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;
bits(128) tmp;
bits(32) tmp2;
tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7):ROL(Vm<95:64>, 7):ROL(Vm<63:32>, 7):ROL(Vm<31:0>, 7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR ROL(tmp2, 23);
result<127:96> = result<127:96> EOR tmp2;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM3SS1

SM3SS1 rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
1	1	0	0	1	1	1	0	0	1	0	Rm	0	Ra	Rn	Rd

Advanced SIMD

```
SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S
```

```
if !HaveChCryptoExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer a = UInt(Ra);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va>	Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) Va = V[a];
Vd<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>), 7);
Vd<95:0> = Zeros();
V[d] = Vd;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM3TT1A

SM3TT1A takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	1	0	0	1	1	1	0	0	1	0	Rm	1	0	imm2	0	0	Rn	Rd

Advanced SIMD

```
SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

```
if !HaveChCryptoExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM3TT1B

SM3TT1B takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	1	0	0	1	1	1	0	0	1	0	Rm	1	0	imm2	0	1	Rn	Rd

Advanced SIMD

```
SM3TT1B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

```
if !HaveChCryptoExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = (Vd<127:96> AND Vd<63:32>) OR (Vd<127:96> AND Vd<95:64>) OR (Vd<63:32> AND Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM3TT2A

SM3TT2A takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	1	0	0	1	1	1	0	0	1	0	Rm	1	0	imm2	1	0	Rn	Rd

Advanced SIMD

```
SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

```
if !HaveChCryptoExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT1;

Wj = Elem[Vm, i, 32];
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM3TT2B

SM3TT2B takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	1	0	0	1	1	1	0	0	1	0	Rm	1	0	imm2	1	1	Rn	Rd

Advanced SIMD

```
SM3TT2B <Vd>.S, <Vn>.S, <Vm>.S[<imm2>]
```

```
if !HaveChCryptoExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer i = UInt(imm2);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
TT2 = (Vd<127:96> AND Vd<95:64>) OR (NOT(Vd<127:96>) AND Vd<63:32>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM4E

SM4 Encode takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	1	Rn					Rd				

Advanced SIMD

```
SM4E <Vd>.4S, <Vn>.4S
```

```
if !HaveChCryptoExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
bits(128) Vn = V[n];
bits(32) intval;
bits(8) sboxout;
bits(128) roundresult;
bits(32) roundkey;
integer index;

roundresult = V[d];
for index = 0 to 3
    roundkey = Elem[Vn, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
V[d] = roundresult;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SM* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
1	1	0	0	1	1	1	0	0	1	1	Rm	1	1	0	0	1	0	Rn	Rd

Advanced SIMD

```
SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S
```

```
if !HaveChCryptoExt() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
bits(128) Vm = V[m];
bits(32) intval;
bits(8) sboxout;
bits(128) result;
bits(32) const;
bits(128) roundresult;
integer index;

roundresult = V[n];
for index = 0 to 3
    const = Elem[Vm, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
V[d] = roundresult;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	0	0	1	Rm					0	Ra					Rn					Rd				
								U								o0															

64-bit

```
SMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
SMULL	`Ra == '11111'`

Operation

```
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, FALSE) + (Int(operand1, FALSE) * Int(operand2, FALSE));

X[d] = result<63:0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAX

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	1	0	0	1	Rn					Rd				
		U																		o1											

Three registers of the same type

```
SMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAXP

Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	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	1	Rn					Rd				
		U																		o1											

Three registers of the same type

```
SMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAXV

Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	1	0	0	0	0	1	0	1	0	1	0	Rn					Rd				
		U													op																

Advanced SIMD

```
SMAXV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMC

Secure Monitor Call causes an exception to EL3.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.

If the values of *HCR_EL2*.TSC and *SCR_EL3*.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in *ESR_ELx*, using the EC value 0x17, that is taken to EL3.

If the value of *HCR_EL2*.TSC is 1, execution of an SMC instruction in a Non-secure EL1 state generates an exception that is taken to EL2, regardless of the value of *SCR_EL3*.SMD. For more information, see *Traps to EL2 of Non-secure EL1 execution of SMC instructions*.

If the value of *HCR_EL2*.TSC is 0 and the value of *SCR_EL3*.SMD is 1, the SMC instruction is UNDEFINED.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	0	imm16																0	0	0	1	1

System

```
SMC #<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

```
AArch64.CheckForSMCUndefOrTrap(imm);

if SCR_EL3.SMD == '1' then
    // SMC disabled
    AArch64.UndefinedFault();
else
    AArch64.CallSecureMonitor(imm);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMIN

Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	1	0	1	1	Rn					Rd				
		U																		o1											

Three registers of the same type

```
SMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMINP

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	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	1	1	Rn					Rd				
		U																		o1											

Three registers of the same type

```
SMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMINV

Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	1	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U													op																

Advanced SIMD

```
SMINV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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														

Vector

```
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 UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLAL` instruction extracts each source vector from the lower half of each source register, while the `SMLAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					1	0	0	0	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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														

Vector

```
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 UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLSL` instruction extracts each source vector from the lower half of each source register, while the `SMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					1	0	1	0	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMNEGL

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of SMSUBL. This means:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	0	0	1	Rm					1	1	1	1	1	1	Rn					Rd				
								U								o0			Ra												

64-bit

```
SMNEGL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMSUBL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMSUBL gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMOV

Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	0	0	0	imm5	0	0	1	0	1	1	Rn	Rd

32-bit (Q == 0)

```
SMOV <Wd>, <Vn>.<Ts>[<index>]
```

64-bit (Q == 1)

```
SMOV <Xd>, <Vn>.<Ts>[<index>]
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when 'xxxxx1' size = 0;     // SMOV [WX]d, Vn.B
    when 'xxxx10' size = 1;     // SMOV [WX]d, Vn.H
    when '1xx100' size = 2;     // SMOV Xd, Vn.S
    otherwise UnallocatedEncoding();

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in “imm5”:

imm5	<Ts>
xxx00	RESERVED
xxxx1	B
xxx10	H

For the 64-bit variant: is an element size specifier, encoded in “imm5”:

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

<index> For the 32-bit variant: is the element index encoded in “imm5”:

imm5	<index>
xxx00	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>

For the 64-bit variant: is the element index encoded in “imm5”:

imm5	<index>
xx000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];

X[d] = SignExtend(Elem[operand, index, esize], datasize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMSUBL

Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	0	0	1	Rm					1	Ra					Rn					Rd				
								U								o0															

64-bit

```
SMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
SMNEGL	`Ra == '11111'`

Operation

```
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, FALSE) - (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d] = result<63:0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	0	1	0	Rm					0	(1)	(1)	(1)	(1)	(1)	Rn					Rd				
								U									Ra														

64-bit

```
SMULH <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
bits(64) operand1 = X[n];
bits(64) operand2 = X[m];

integer result;

result = Int(operand1, FALSE) * Int(operand2, FALSE);

X[d] = result<127:64>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMULL` instruction extracts vector elements from the lower half of the first source register, while the `SMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size		L	M	Rm				1	0	1	0	H	0	Rn					Rd				
		U																													

Vector

```
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 UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULL, SMULL2 (vector)

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The `SMULL` instruction extracts each source vector from the lower half of each source register, while the `SMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					1	1	0	0	0	0	Rn					Rd				
		U																													

Three registers, not all the same type

```
SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULL

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of SMADDL. This means:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	0	0	1	Rm					0	1	1	1	1	1	Rn					Rd				
								U								o0			Ra												

64-bit

```
SMULL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMADDL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQABS

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	0	0	0	0	0	0	1	1	1	1	0	Rn					Rd				
		U																													

Scalar

```
SQABS <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	0	1	1	1	1	0	Rn					Rd				
		U																													

Vector

```
SQABS <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQADD

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					0	0	0	0	1	1	Rn					Rd				
		U																													

Scalar

```
SQADD <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	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	1	1	Rn					Rd				
		U																													

Vector

```
SQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLAL, SQDMLAL2 (by element)

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQDMLAL` instruction extracts vector elements from the lower half of the first source register, while the `SQDMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size		L	M	Rm				0	0	1	1	H	0	Rn					Rd				
																	o2														

Scalar

```
SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size		L	M	Rm				0	0	1	1	H	0	Rn					Rd				
																	o2														

Vector

```
SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(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;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQDMLAL` instruction extracts each source vector from the lower half of each source register, while the `SQDMLAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					1	0	0	1	0	0	Rn					Rd				
																		o1													

Scalar

```
SQDMLAL <Va><d>, <Vb><n>, <Vb><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
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													

Vector

```
SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLSL, SQDMLSL2 (by element)

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQDMLSL` instruction extracts vector elements from the lower half of the first source register, while the `SQDMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size		L	M	Rm				0	1	1	1	H	0	Rn					Rd				
																	o2														

Scalar

```
SQDMLSL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size		L	M	Rm				0	1	1	1	H	0	Rn					Rd				
																	o2														

Vector

```
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 UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(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;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLSL, SQDMLSL2 (vector)

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQDMLSL` instruction extracts each source vector from the lower half of each source register, while the `SQDMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					1	0	1	1	0	0	Rn					Rd				
																		o1													

Scalar

```
SQDMLSL <Va><d>, <Vb><n>, <Vb><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
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													

Vector

```
SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMULH (by element)

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see *SQRDMULH*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size		L	M	Rm				1	1	0	0	H	0	Rn					Rd				
																			op												

Scalar

```
SQDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size		L	M	Rm				1	1	0	0	H	0	Rn					Rd				
																			op												

Vector

```
SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMULH (vector)

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see *SQRDMULH*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					1	0	1	1	0	1	Rn					Rd				
		U																													

Scalar

```
SQDMULH <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					1	0	1	1	0	1	Rn					Rd				
		U																													

Vector

```
SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQDMULL` instruction extracts the first source vector from the lower half of the first source register, while the `SQDMULL2` instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size		L	M	Rm				1	0	1	1	H	0	Rn					Rd				

Scalar

```
SQDMULL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size		L	M	Rm				1	0	1	1	H	0	Rn					Rd				

Vector

```
SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <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;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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				

Scalar

```
SQDMULL <Va><d>, <Vb><n>, <Vb><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					1	1	0	1	0	0	Rn					Rd				

Vector

```
SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQNEG

Signed saturating Negate. This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	0	0	0	0	0	0	1	1	1	1	0	Rn					Rd				
		U																													

Scalar

```
SQNEG <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	0	1	1	1	1	0	Rn					Rd				
		U																													

Vector

```
SQNEG <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMLAH (by element)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSR*.QC, is set if saturation occurs.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	size		L	M	Rm				1	1	0	1	H	0	Rn					Rd				
																		S													

Scalar

```
SQRDMLAH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

```
if !HaveQRDMLAHExt() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				1	1	0	1	H	0	Rn					Rd				
																		S													

Vector

```
SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
if !HaveQRDMLAHExt() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSR*.QC, is set if saturation occurs.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		0	Rm					1	0	0	0	0	1	Rn					Rd				
																				S											

Scalar

```
SQRDMLAH <V><d>, <V><n>, <V><m>
```

```
if !HaveQRDMLAHExt() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		0	Rm					1	0	0	0	0	1	Rn					Rd				
																				S											

Vector

```
SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveQRDMLAHExt() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMLSH (by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSR*.QC, is set if saturation occurs.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	size		L	M	Rm				1	1	1	1	H	0	Rn					Rd				
																		S													

Scalar

```
SQRDMLSH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

```
if !HaveQRDMLAHExt() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				1	1	1	1	H	0	Rn					Rd				
																		S													

Vector

```
SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
if !HaveQRDMLAHExt() then UnallocatedEncoding();

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMLSH (vector)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSR*.QC, is set if saturation occurs.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		0	Rm					1	0	0	0	1	1	Rn					Rd				
																				S											

Scalar

```
SQRDMLSH <V><d>, <V><n>, <V><m>
```

```
if !HaveQRDMLAHExt() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		0	Rm					1	0	0	0	1	1	Rn					Rd				
																				S											

Vector

```
SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
if !HaveQRDMLAHExt() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMULH (by element)

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see *SQDMULH*.

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size		L	M	Rm				1	1	0	1	H	0	Rn					Rd				
																			op												

Scalar

```
SQRDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size		L	M	Rm				1	1	0	1	H	0	Rn					Rd				
																			op												

Vector

```
SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMULH (vector)

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see *SQDMULH*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	Rm					1	0	1	1	0	1	Rn					Rd				
		U																													

Scalar

```
SQRDMULH <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					1	0	1	1	0	1	Rn					Rd				
		U																													

Vector

```
SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRSHL

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see *SQSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					0	1	0	1	1	1	Rn					Rd				
		U																	R	S											

Scalar

```
SQRSHL <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	0	1	1	1	Rn					Rd				
		U																	R	S											

Vector

```
SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRSHRN, SQRSHRN2

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see *SQSHRN*.

The `SQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	0	0	1	1	1	Rn					Rd				
		U							immh											op											

Scalar

```
SQRSHRN <Vb><d>, <Va><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	1	1	Rn					Rd				
		U							immh											op											

Vector

```
SQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRSHRUN, SQRSHRUN2

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see *SQSHRUN*.

The `SQRSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQRSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	0	0	0	1	1	Rn					Rd				
									immh											op										

Scalar

```
SQRSHRUN <Vb><d>, <Va><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	0	1	1	Rn					Rd				
									immh											op										

Vector

```
SQRSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSHL (immediate)

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see *UQRSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	1	1	1	0	1	Rn					Rd				
		U							immh										op												

Scalar

```
SQSHL <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	1	1	1	0	1	Rn					Rd				
		U							immh										op												

Vector

```
SQSHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSHL (register)

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see *SQRSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					0	1	0	0	1	1	Rn					Rd				
		U																	R	S											

Scalar

```
SQSHL <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	0	0	1	1	Rn					Rd				
		U																	R	S											

Vector

```
SQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSHLU

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see *UQRSHL*.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	1	1	0	0	1	Rn					Rd				
		U							immh										op												

Scalar

```
SQSHLU <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	1	0	0	1	Rn					Rd				
		U							immh										op												

Vector

```
SQSHLU <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSHRN, SQSHRN2

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see *SQRSHRN*.

The `SQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn					Rd				
		U							immh											op											

Scalar

```
SQSHRN <Vb><d>, <Va><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn					Rd				
		U							immh											op											

Vector

```
SQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSHRUN, SQSHRUN2

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see *SQRSHRUN*.

The `SQSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	0	0	0	0	1	Rn					Rd				
									immh											op										

Scalar

```
SQSHRUN <Vb><d>, <Va><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	0	0	1	Rn					Rd				
									immh											op										

Vector

```
SQSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSUB

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set. Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					0	0	1	0	1	1	Rn					Rd				
		U																													

Scalar

```
SQSUB <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	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	1	1	Rn					Rd				
		U																													

Vector

```
SQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQXTN, SQXTN2

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	0	0	0	0	1	0	1	0	0	1	0	Rn					Rd				
		U																													

Scalar

```
SQXTN <Vb><d>, <Va><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	1	0	1	0	0	1	0	Rn					Rd				
		U																													

Vector

```
SQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQXTUN, SQXTUN2

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQXTUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	0	0	0	0	1	0	0	1	0	1	0	Rn					Rd				

Scalar

```
SQXTUN <Vb><d>, <Va><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	1	0	0	1	0	1	0	Rn					Rd				

Vector

```
SQXTUN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRHADD

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see *SHADD*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	0	0	1	0	1	Rn					Rd				
		U																													

Three registers of the same type

```
SRHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1+element2+1)<esize:1>;

V[d] = result;
```

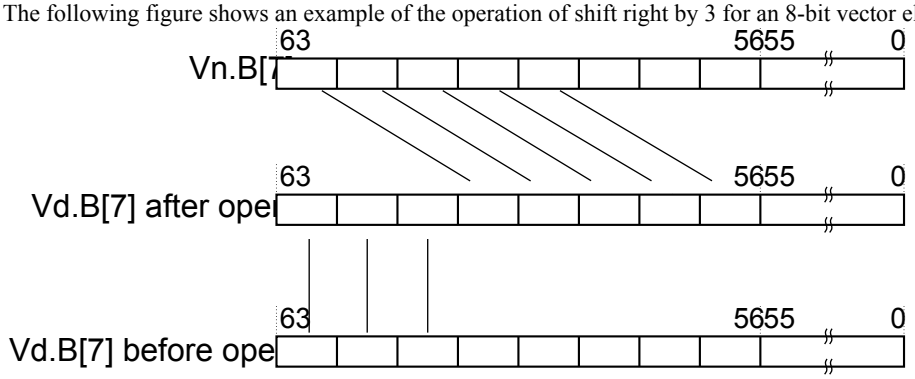
Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRI

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

The following figure shows an example of the operation of shift right by 3 for an 8-bit vector element.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	1	0	0	0	1	Rn					Rd				
									immh																						

Scalar

```
SRI <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	0	0	0	1	Rn					Rd				
									immh																						

Vector

```
SRI <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
bits(esize) mask = LSR(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSR(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRSHL

Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see *SSHL*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					0	1	0	1	0	1	Rn					Rd				
		U																	R	S											

Scalar

```
SRSHL <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	0	1	0	1	Rn					Rd				
		U																	R	S											

Vector

```
SRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRSHR

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see *SSHR*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	0	1	0	0	1	Rn					Rd				
		U							immh									o1	o0												

Scalar

```
SRSHR <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	0	1	0	0	1	Rn					Rd				
		U							immh									o1	o0												

Vector

```
SRSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRSRA

Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see *SSRA*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	0	1	1	0	1	Rn					Rd				
		U							immh									o1	o0												

Scalar

```
SRSRA <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	0	1	1	0	1	Rn					Rd				
		U							immh									o1	o0												

Vector

```
SRSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSHL

Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see *SRSHL*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	Rm					0	1	0	0	0	1	Rn					Rd				
		U																	R	S											

Scalar

```
SSHL <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	0	0	0	1	Rn					Rd				
		U																	R	S											

Vector

```
SSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSHLL, SSHLL2

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SSHLL` instruction extracts vector elements from the lower half of the source register, while the `SSHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias SXTL, SXTL2.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	1	0	0	1	Rn					Rd				
		U							immh																						

Vector

```
SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	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;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSHR

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see *SRSHR*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	0	0	0	0	1	Rn					Rd				
		U							immh									o1	o0												

Scalar

```
SSHR <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	0	0	0	0	1	Rn					Rd				
		U							immh									o1	o0												

Vector

```
SSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSRA

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see *SRSRA*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	0	0	1	0	1	Rn					Rd				
		U							immh									o1	o0												

Scalar

```
SSRA <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	0	0	1	0	1	Rn					Rd				
		U							immh									o1	o0												

Vector

```
SSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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													

Three registers, not all the same type

```
SSUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSUBW, SSUBW2

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The `SSUBW` instruction extracts the second source vector from the lower half of the second source register, while the `SSUBW2` instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	0	1	1	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
SSUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	x	x	1	x	size		Rn					Rt				
									L							opcode															

One register (opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>]
```

Two registers (opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Three registers (opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Four registers (opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm					x	x	1	x	size		Rn					Rt				
									L							opcode															

One register, immediate offset (Rm == 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory. Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	0	S	size		Rn					Rt				
									L	R						opcode															

8-bit (opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	0	Rm					x	x	0	S	size		Rn					Rt				
									L	R						opcode															

8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	size		Rn					Rt				
									L							opcode															

No offset

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm					1	0	0	0	size		Rn					Rt				
									L							opcode															

Immediate offset (Rm == 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	0	S	size		Rn					Rt				
									L	R						opcode															

8-bit (opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	1	Rm					x	x	0	S	size		Rn					Rt				
									L	R						opcode															

8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	size		Rn					Rt				
									L							opcode															

No offset

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm					0	1	0	0	size		Rn					Rt				
									L							opcode															

Immediate offset (Rm == 11111)

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	1	S	size		Rn					Rt				
									L	R						opcode															

8-bit (opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	0	Rm					x	x	1	S	size		Rn					Rt				
									L	R						opcode															

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	size		Rn					Rt				
									L							opcode															

No offset

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm					0	0	0	0	size		Rn					Rt				
									L							opcode															

Immediate offset (Rm == 11111)

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UnallocatedEncoding();

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then ReservedValue();
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer e, r, s, tt;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST4 (single structure)

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	1	S	size		Rn					Rt				
									L	R						opcode															

8-bit (opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	1	Rm					x	x	1	S	size		Rn					Rt				
									L	R						opcode															

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UnallocatedEncoding();
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UnallocatedEncoding();
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UnallocatedEncoding();
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UnallocatedEncoding();
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
integer s;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADD has no memory ordering semantics.
- STADDL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	0	0	0	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

32-bit, no memory ordering (size == 10 && R == 0)

```
STADD <Ws>, [<Xn|SP>]
```

32-bit, release (size == 10 && R == 1)

```
STADDL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && R == 0)

```
STADD <Xs>, [<Xn|SP>]
```

64-bit, release (size == 11 && R == 1)

```
STADDL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = data + value;
Mem[address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDB has no memory ordering semantics.
- STADDLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	0	0	0	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STADDB <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STADDLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, AccType_ATOMICRW];

result = data + value;
Mem[address, 1, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH has no memory ordering semantics.
- STADDLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs					0	0	0	0	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STADDH <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STADDLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, AccType_ATOMICRW];

result = data + value;
Mem[address, 2, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR has no memory ordering semantics.
- STCLRL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	0	0	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

32-bit, no memory ordering (size == 10 && R == 0)

```
STCLR <Ws>, [<Xn|SP>]
```

32-bit, release (size == 10 && R == 1)

```
STCLRL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && R == 0)

```
STCLR <Xs>, [<Xn|SP>]
```

64-bit, release (size == 11 && R == 1)

```
STCLRL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = data AND NOT(value);
Mem[address, datasize DIV 8, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRB has no memory ordering semantics.
- STCLRLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	0	0	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STCLRB <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STCLRLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, AccType_ATOMICRW];

result = data AND NOT(value);
Mem[address, 1, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRH has no memory ordering semantics.
- STCLRLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs					0	0	0	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STCLRH <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STCLRLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, AccType_ATOMICRW];

result = data AND NOT(value);
Mem[address, 2, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STEOR, STEORL

Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEOR has no memory ordering semantics.
- STEORL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	0	1	0	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

32-bit, no memory ordering (size == 10 && R == 0)

```
STEOR <Ws>, [<Xn|SP>]
```

32-bit, release (size == 10 && R == 1)

```
STEORL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && R == 0)

```
STEOR <Xs>, [<Xn|SP>]
```

64-bit, release (size == 11 && R == 1)

```
STEORL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = data EOR value;
Mem[address, datasize DIV 8, stacctype] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORB has no memory ordering semantics.
- STEORLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	0	1	0	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STEORB <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STEORLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, AccType_ATOMICRW];

result = data EOR value;
Mem[address, 1, stacctype] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORH has no memory ordering semantics.
- STEORLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs	0	0	1	0	0	0	Rn	1	1	1	1	1
size					V			A				o3	opc										

No memory ordering (R == 0)

```
STEORH <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STEORLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, AccType_ATOMICRW];

result = data EOR value;
Mem[address, 2, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

32-bit (size == 10)

```
STLLR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STLLR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, dbytes, AccType_LIMITEDORDERED] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
STLLRB <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, 1, AccType_LIMITEDORDERED] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
STLLRH <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, 2, AccType_LIMITEDORDERED] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L				Rs			o0			Rt2												

32-bit (size == 10)

```
STLR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STLR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, dbytes, AccType_ORDERED] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
STLRB <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, 1, AccType_ORDERED] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

No offset

```
STLRH <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];
Mem[address, 2, AccType_ORDERED] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
1	sz	0	0	1	0	0	0	0	0	1	Rs	1	Rt2	Rn	Rt
									L			o0			

32-bit (sz == 0)

```
STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit (sz == 1)

```
STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
```

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.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || (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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if 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_ORDERED] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
```

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.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t 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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = 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_ORDERED] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	Rs					1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L							o0	Rt2														

No offset

```
STLXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release
```

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.

Operation

```
bits(64) address;
bits(8) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t 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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ORDERED] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	Rs					1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L							o0		Rt2													

No offset

```
STLXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release
```

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.

Operation

```
bits(64) address;
bits(16) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t 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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ORDERED] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STNP (SIMD&FP)

Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 28 27	26	25 24 23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
opc	1 0 1	1	0 0 0	0	imm7	Rt2	Rn	Rt
				L				

32-bit (opc == 00)

```
STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 01)

```
STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit (opc == 10)

```
STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, AccType_VECSTREAM] = data1;
Mem[address+dbytes, dbytes, AccType_VECSTREAM] = data2;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair*.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
x	0	1	0	1	0	0	0	0	0	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data1 = X[t];
data2 = X[t2];
Mem[address, dbytes, AccType_STREAM] = data1;
Mem[address+dbytes, dbytes, AccType_STREAM] = data2;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	0	1	0	imm7							Rt2					Rn						Rt			
									L																						

32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 01)

```
STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	1	1	0	imm7							Rt2					Rn						Rt			
									L																						

32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 01)

```
STP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	1	0	1	0	0	imm7							Rt2					Rn						Rt			
									L																						

32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 01)

```
STP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16. For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, AccType_VEC] = data1;
Mem[address+dbytes, dbytes, AccType_VEC] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STP

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
x	0	1	0	1	0	0	0	1	0	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
x	0	1	0	1	0	0	1	1	0	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
x	0	1	0	1	0	0	1	0	0	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2];
Mem[address, dbytes, AccType_NORMAL] = data1;
Mem[address+dbytes, dbytes, AccType_NORMAL] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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>], #<simm>
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>], #<simm>
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>], #<simm>
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>], #<simm>
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	0	0	imm9									1	1	Rn					Rt				
								opc																							

8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>, #<simm>]!
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, #<simm>]!
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>, #<simm>]!
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
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
size	1 1 1	1	0 1	x 0	imm12	Rn	Rt
				opc			

8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	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;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	x	1	1	1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
size								opc							

32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>], #<simm>
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
1	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>, #<simm>]!
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
1	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.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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-bit (size == 00 && opc == 00 && option != 011)

```
STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

8-bit (size == 00 && opc == 00 && option == 011)

```
STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

128-bit (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 UnallocatedEncoding();
if option<1> == '0' then UnallocatedEncoding();     // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	For the 8-bit variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	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;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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>}}]
```

```
integer scale = UInt(size);
if option<1> == '0' then UnallocatedEncoding();     // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	0	0	imm9									0	1	Rn					Rt				
size								opc																							

Post-index

```
STRB <Wt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn					Rt				
size								opc																							

Pre-index

```
STRB <Wt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	1	0	0	imm12												Rn					Rt				
size								opc																							

Unsigned offset

```
STRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

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);
```

Operation

```
bits(64) address;
bits(8) data;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];
Mem[address, 1, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	0	1	Rm					option			S	1	0	Rn					Rt				
size								opc																							

Extended register (option != 011)

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register (option == 011)

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then UnallocatedEncoding();     // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

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	SXTX

<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);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 1, AccType_NORMAL] = data;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	0	0	imm9									0	1	Rn					Rt				
size								opc																							

Post-index

```
STRH <Wt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn					Rt				
size								opc																							

Pre-index

```
STRH <Wt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
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																							

Unsigned offset

```
STRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

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);
```

Operation

```
bits(64) address;
bits(16) data;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];
Mem[address, 2, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																							

32-bit

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UnallocatedEncoding();     // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 2, AccType_NORMAL] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSET` has no memory ordering semantics.
- `STSETL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	0	1	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

32-bit, no memory ordering (size == 10 && R == 0)

```
STSET <Ws>, [<Xn|SP>]
```

32-bit, release (size == 10 && R == 1)

```
STSETL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && R == 0)

```
STSET <Xs>, [<Xn|SP>]
```

64-bit, release (size == 11 && R == 1)

```
STSETL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = data OR value;
Mem[address, datasize DIV 8, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETB has no memory ordering semantics.
- STSETLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	0	1	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STSETB <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STSETLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, AccType_ATOMICRW];

result = data OR value;
Mem[address, 1, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETH has no memory ordering semantics.
- STSETLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs					0	0	1	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STSETH <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STSETLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, AccType_ATOMICRW];

result = data OR value;
Mem[address, 2, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAX` has no memory ordering semantics.
- `STSMAXL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	1	0	0	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

32-bit, no memory ordering (size == 10 && R == 0)

```
STSMAX <Ws>, [<Xn|SP>]
```

32-bit, release (size == 10 && R == 1)

```
STSMAXL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && R == 0)

```
STSMAX <Xs>, [<Xn|SP>]
```

64-bit, release (size == 11 && R == 1)

```
STSMAXL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = if SInt(data) > SInt(value) then data else value;
Mem[address, datasize DIV 8, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSMAXB, STSMAXLB

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXB has no memory ordering semantics.
- STSMAXLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs	0	1	0	0	0	0	Rn	1	1	1	1	1
size					V			A				o3	opc										

No memory ordering (R == 0)

```
STSMAXB <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STSMAXLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, AccType_ATOMICRW];

result = if SInt(data) > SInt(value) then data else value;
Mem[address, 1, stacctype] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXH` has no memory ordering semantics.
- `STSMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs	0	1	0	0	0	0	Rn	1	1	1	1	1
size					V			A				o3	opc										

No memory ordering (R == 0)

```
STSMAXH <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STSMAXLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, AccType_ATOMICRW];

result = if SInt(data) > SInt(value) then data else value;
Mem[address, 2, stacctype] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMIN` has no memory ordering semantics.
- `STSMINL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	1	0	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

32-bit, no memory ordering (size == 10 && R == 0)

```
STSMIN <Ws>, [<Xn|SP>]
```

32-bit, release (size == 10 && R == 1)

```
STSMINL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && R == 0)

```
STSMIN <Xs>, [<Xn|SP>]
```

64-bit, release (size == 11 && R == 1)

```
STSMINL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = if SInt(data) > SInt(value) then value else data;
Mem[address, datasize DIV 8, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINB` has no memory ordering semantics.
- `STSMINLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	1	0	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STSMINB <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STSMINLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, AccType_ATOMICRW];

result = if SInt(data) > SInt(value) then value else data;
Mem[address, 1, stacctype] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINH has no memory ordering semantics.
- STSMINLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs					0	1	0	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STSMINH <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STSMINLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, AccType_ATOMICRW];

result = if SInt(data) > SInt(value) then value else data;
Mem[address, 2, stacctype] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for 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>{, #<simm>}]
```

64-bit (size == 11)

```
STTR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
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_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    acctype = AccType_NORMAL;

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, acctype] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for 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																							

Unscaled offset

```
STTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

```
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_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    acctype = AccType_NORMAL;
```

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 1, acctype] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for 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					

Unscaled offset

```
STTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

```
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_UNPRIV;
if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1) then
    acctype = AccType_NORMAL;
```

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 2, acctype] = data;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAX has no memory ordering semantics.
- STUMAXL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	1	1	0	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

32-bit, no memory ordering (size == 10 && R == 0)

```
STUMAX <Ws>, [<Xn|SP>]
```

32-bit, release (size == 10 && R == 1)

```
STUMAXL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && R == 0)

```
STUMAX <Xs>, [<Xn|SP>]
```

64-bit, release (size == 11 && R == 1)

```
STUMAXL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = if UInt(data) > UInt(value) then data else value;
Mem[address, datasize DIV 8, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXB has no memory ordering semantics.
- STUMAXLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	1	1	0	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STUMAXB <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STUMAXLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, AccType_ATOMICRW];

result = if UInt(data) > UInt(value) then data else value;
Mem[address, 1, stacctype] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXH has no memory ordering semantics.
- STUMAXLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs					0	1	1	0	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STUMAXH <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STUMAXLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, AccType_ATOMICRW];

result = if UInt(data) > UInt(value) then data else value;
Mem[address, 2, stacctype] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMIN` has no memory ordering semantics.
- `STUMINL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs					0	1	1	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

32-bit, no memory ordering (size == 10 && R == 0)

```
STUMIN <Ws>, [<Xn|SP>]
```

32-bit, release (size == 10 && R == 1)

```
STUMINL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && R == 0)

```
STUMIN <Xs>, [<Xn|SP>]
```

64-bit, release (size == 11 && R == 1)

```
STUMINL <Xs>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = if UInt(data) > UInt(value) then value else data;

Mem[address, datasize DIV 8, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINB` has no memory ordering semantics.
- `STUMINLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs					0	1	1	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STUMINB <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STUMINLB <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 1, AccType_ATOMICRW];

result = if UInt(data) > UInt(value) then value else data;

Mem[address, 1, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINH has no memory ordering semantics.
- STUMINLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs					0	1	1	1	0	0	Rn					1	1	1	1	1
size					V			A								o3	opc														

No memory ordering (R == 0)

```
STUMINH <Ws>, [<Xn|SP>]
```

Release (R == 1)

```
STUMINLH <Ws>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// All observers in the shareability domain observe the
// following load and store atomically.
data = Mem[address, 2, AccType_ATOMICRW];

result = if UInt(data) > UInt(value) then value else data;

Mem[address, 2, stacctype] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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>{, #<simm>}]
```

16-bit (size == 01 && opc == 00)

```
STUR <Ht>, [<Xn|SP>{, #<simm>}]
```

32-bit (size == 10 && opc == 00)

```
STUR <St>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11 && opc == 00)

```
STUR <Dt>, [<Xn|SP>{, #<simm>}]
```

128-bit (size == 00 && opc == 10)

```
STUR <Qt>, [<Xn|SP>{, #<simm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UnallocatedEncoding();
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<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_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
```

Operation

```
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	0	0	imm9									0	0	Rn					Rt				
size								opc																							

Unscaled offset

```
STURB <Wt>, [<Xn|SP>{, #<simm>}]
```

```
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);
```

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 1, AccType_NORMAL] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset

```
STURH <Wt>, [<Xn|SP>{, #<simm>}]
```

```
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);
```

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 2, AccType_NORMAL] = data;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STXP

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	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

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
```

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.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || (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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if 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_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	0	0	Rs					0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L							o0			Rt2												

32-bit (size == 10)

```
STXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
```

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.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t 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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = 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_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	Rs					0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L							o0	Rt2														

No offset

```
STXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);     // ignored by all loads and store-release
```

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.

Operation

```
bits(64) address;
bits(8) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t 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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	Rs					0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L							o0	Rt2														

No offset

```
STXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);     // ignored by all loads and store-release
```

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.

Operation

```
bits(64) address;
bits(16) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t 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   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13	12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1	0	0	1	0	1	1	0	0	1	Rm	option	imm3	Rn	Rd
	op	S													

32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (immediate)

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23 22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1	0	1	0	0	0	1	shift	imm12	Rn	Rd
	op	S									

32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12:Zeros(12), datasize);
    when '1x' ReservedValue();
```

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<imm> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift”:

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;

operand2 = NOT(imm);
(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias NEG (shifted register).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	0	1	0	1	1	shift		0	Rm					imm6						Rn					Rd				
	op	S																													

32-bit (sf == 0)

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

Alias	Is preferred when
NEG (shifted register)	`Rn == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (vector)

Subtract (vector). This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	Rm					1	0	0	0	0	1	Rn					Rd				
		U																													

Scalar

```
SUB <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					1	0	0	0	0	1	Rn					Rd				
		U																													

Vector

```
SUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUBHN, SUBHN2

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see *RSUBHN*.

The `SUBHN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SUBHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	Rm					0	1	1	0	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
SUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUBS (extended register)

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMP (extended register).

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13	12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1	1	0	1	0	1	1	0	0	1	Rm	option	imm3	Rn	Rd
	op	S													

32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Alias Conditions

Alias	Is preferred when
CMP (extended register)	`Rd == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUBS (immediate)

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMP (immediate).

31	30	29	28	27	26	25	24	23 22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1	1	1	0	0	0	1	shift	imm12	Rn	Rd
	op	S									

32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12:Zeros(12), datasize);
    when '1x' ReservedValue();
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn|WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift>	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “shift”:

shift	<shift>
00	LSL #0
01	LSL #12
1x	RESERVED

Alias Conditions

Alias	Is preferred when
CMP (immediate)	`Rd == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;
bits(4) nzcv;

operand2 = NOT(imm);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUBS (shifted register)

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases CMP (shifted register), and NEGS.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	0	1	0	1	1	shift		0	Rm					imm6						Rn					Rd				
	op	S																													

32-bit (sf == 0)

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if shift == '11' then ReservedValue();
if sf == '0' && imm6<5> == '1' then ReservedValue();

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

Alias	Is preferred when
CMP (shifted register)	`Rd == '11111'`
NEGS	`Rn == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUQADD

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	0	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
		U																													

Scalar

```
SUQADD <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
		U																													

Vector

```
SUQADD <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SVC

Supervisor Call causes an exception to be taken to EL1.

On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in *ESR_ELx*, using the EC value 0x15, and the value of the immediate argument.

31	30	29	28	27	26	25	24	23	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	0	1

System

```
SVC #<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

```
AArch64.CallSupervisor(imm);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `SWPA` and `SWPAL` load from memory with acquire semantics.
- `SWPL` and `SWPAL` store to memory with release semantics.
- `SWP` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs					1	0	0	0	0	0	Rn					Rt				
size																															

32-bit, acquire (size == 10 && A == 1 && R == 0)

```
SWPA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release (size == 10 && A == 1 && R == 1)

```
SWPAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering (size == 10 && A == 0 && R == 0)

```
SWP <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release (size == 10 && A == 0 && R == 1)

```
SWPL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire (size == 11 && A == 1 && R == 0)

```
SWPA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release (size == 11 && A == 1 && R == 1)

```
SWPAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering (size == 11 && A == 0 && R == 0)

```
SWP <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release (size == 11 && A == 0 && R == 1)

```
SWPL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

// all observers in the shareability domain
// observe the load and store atomically
Mem[address, datasize DIV 8, stacctype] = X[s];

X[t] = ZeroExtend(data, regsize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs					1	0	0	0	0	0	Rn					Rt				
size																															

Acquire (A == 1 && R == 0)

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0)

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1)

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];

// all observers in the shareability domain
// observe the load and store atomically
Mem[address, 1, stacctype] = X[s];

X[t] = ZeroExtend(data, 32);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `SWPAH` and `SWPALH` load from memory with acquire semantics.
- `SWPLH` and `SWPALH` store to memory with release semantics.
- `SWPH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(ARMv8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs					1	0	0	0	0	0	Rn					Rt				
size																															

Acquire (A == 1 && R == 0)

```
SWPAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release (A == 1 && R == 1)

```
SWPALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering (A == 0 && R == 0)

```
SWPH <Ws>, <Wt>, [<Xn|SP>]
```

Release (A == 0 && R == 1)

```
SWPLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UnallocatedEncoding();

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];

// all observers in the shareability domain
// observe the load and store atomically
Mem[address, 2, stacctype] = X[s];

X[t] = ZeroExtend(data, 32);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	1	1	0	N	0	0	0	0	0	0	0	0	0	1	1	1	Rn					Rd				
	opc											immr						imms													

32-bit (sf == 0 && N == 0)

```
SXTB <Wd>, <Wn>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

```
SXTB <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #7
```

and is always the preferred disassembly.

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.
<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

The description of SBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	1	1	0	N	0	0	0	0	0	0	0	0	1	1	1	1	Rn					Rd				
	opc											immr						imms													

32-bit (sf == 0 && N == 0)

```
SXTH <Wd>, <Wn>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

```
SXTH <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #15
```

and is always the preferred disassembly.

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.
<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

The description of SBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																		

Vector

```
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.

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	1	1	1	1	1	Rn					Rd				
sf	opc								N			immr						imms													

64-bit

```
SXTW <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #31
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<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

The description of SBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SYS

System instruction. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.

This instruction is used by the aliases AT, DC, IC, and TLBI.

31	30	29	28	27	26	25	24	23	22	21	20	19	18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3 2 1 0
1	1	0	1	0	1	0	1	0	0	0	0	1	op1	CRn	CRm	op2	Rt
										L							

System

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<Xt>	Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
AT	`CRn == '0111' && CRm == '100x' && SysOp(op1,'0111',CRm,op2) == Sys_AT`
DC	`CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_DC`
IC	`CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_IC`
TLBI	`CRn == '1000' && SysOp(op1,'1000',CRm,op2) == Sys_TLBI`

Operation

```
AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SYSL

System instruction with result. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3 2 1 0
1	1	0	1	0	1	0	1	0	0	1	0	1	op1	CRn	CRm	op2	Rt
										L							

System

```
SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

```
X[t] = AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys_crm, sys_op2);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBL

Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	0	0	0	Rm	0	len	0	0	0	Rn	Rd
														op				

Two register table (len == 01)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table (len == 10)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table (len == 11)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table (len == 00)

```
TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;
integer i;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
b5	0	1	1	0	1	1	1	b40					imm14														Rt				
							op																								

14-bit signed PC-relative branch offset

```
TBNZ <R><t>, #<imm>, <label>
```

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

<R> Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBX

Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14 13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	0	0	0	Rm	0	len	1	0	0	Rn	Rd
														op				

Two register table (len == 01)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table (len == 10)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table (len == 11)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table (len == 00)

```
TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;
integer i;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

31	30	29	28	27	26	25	24	23 22 21 20 19	18 17 16 15 14 13 12 11 10 9 8 7 6 5	4 3 2 1 0
b5	0	1	1	0	1	1	0	b40	imm14	Rt
							op			

14-bit signed PC-relative branch offset

```
TBZ <R><t>, #<imm>, <label>
```

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

<R> Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t];

if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TLBI

TLB Invalidate operation. For more information, see *A64 system instructions for TLB maintenance*.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	1	op1			1	0	0	0	CRm				op2			Rt				
										L						CRn															

System

```
TLBI <tlbi_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, C8, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1,'1000',CRm,op2) == Sys_TLBI`.

Assembler Symbols

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<tlbi_op> Is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in “op1:CRm:op2”:

op1	CRm	op2	<tlbi_op>
000	0011	000	VMALLE1IS
000	0011	001	VAE1IS
000	0011	010	ASIDE1IS
000	0011	011	VAAE1IS
000	0011	101	VALE1IS
000	0011	111	VAALE1IS
000	0111	000	VMALLE1
000	0111	001	VAE1
000	0111	010	ASIDE1
000	0111	011	VAAE1
000	0111	101	VALE1
000	0111	111	VAALE1
100	0000	001	IPAS2E1IS
100	0000	101	IPAS2LE1IS
100	0011	000	ALLE2IS
100	0011	001	VAE2IS
100	0011	100	ALLE1IS
100	0011	101	VALE2IS
100	0011	110	VMALLS12E1IS
100	0100	001	IPAS2E1
100	0100	101	IPAS2LE1
100	0111	000	ALLE2
100	0111	001	VAE2
100	0111	100	ALLE1
100	0111	101	VALE2
100	0111	110	VMALLS12E1
110	0011	000	ALLE3IS
110	0011	001	VAE3IS
110	0011	101	VALE3IS
110	0111	000	ALLE3
110	0111	001	VAE3
110	0111	101	VALE3

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

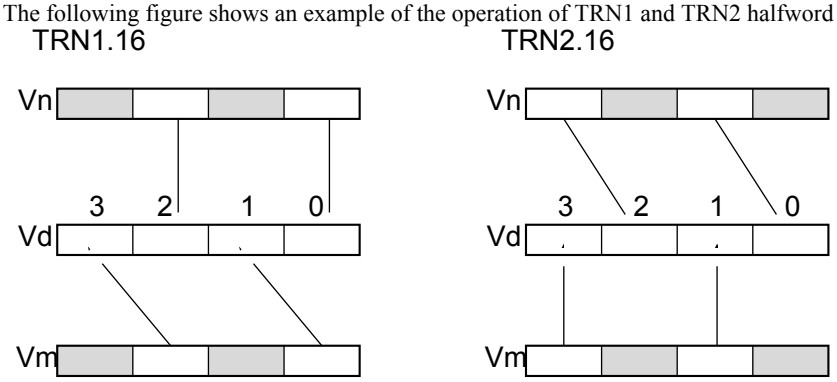
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TRN1

Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

By using this instruction with `TRN2`, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		0	Rm					0	0	1	0	1	0	Rn					Rd				
																	op														

Advanced SIMD

```
TRN1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer p;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

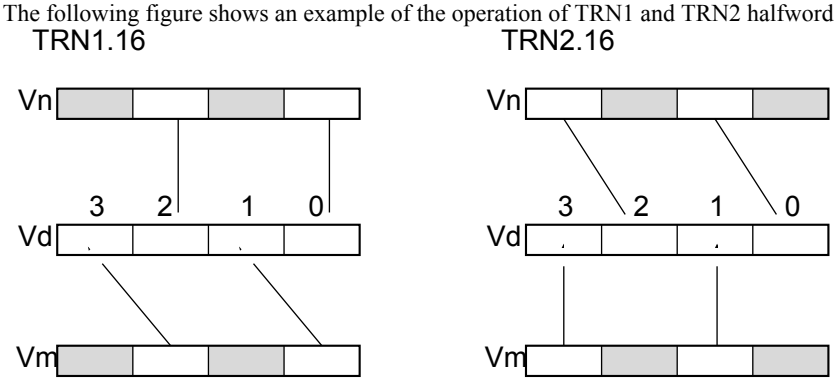
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TRN2

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

By using this instruction with `TRN1`, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		0	Rm					0	1	1	0	1	0	Rn					Rd				
																	op														

Advanced SIMD

```
TRN2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer p;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TST (immediate)

Test bits (immediate), setting the condition flags and discarding the result: `Rn AND imm`.

This is an alias of ANDS (immediate). This means:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1	1	1	0	0	1	0	0	N	immr	imms	Rn	1 1 1 1 1
	opc												Rd

32-bit (sf == 0 && N == 0)

```
TST <Wn>, #<imm>
```

is equivalent to

```
ANDS WZR, <Wn>, #<imm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
TST <Xn>, #<imm>
```

is equivalent to

```
ANDS XZR, <Xn>, #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

The description of ANDS (immediate) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TST (shifted register)

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ANDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	0	1	0	1	0	shift		0	Rm					imm6						Rn					1	1	1	1	1
	opc									N																			Rd		

32-bit (sf == 0)

```
TST <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
TST <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABA

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	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	1	1	Rn					Rd				
		U																		ac											

Three registers of the same type

```
UABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABAL` instruction extracts each source vector from the lower half of each source register, while the `UABAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	1	0	1	0	0	Rn					Rd				
		U																op													

Three registers, not all the same type

```
UABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABD

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	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	1	Rn					Rd				
		U																		ac											

Three registers of the same type

```
UABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABDL, UABDL2

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABDL` instruction extracts each source vector from the lower half of each source register, while the `UABDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	1	1	1	0	0	Rn					Rd				
		U																op													

Three registers, not all the same type

```
UABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADALP

Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	0	1	1	0	1	0	Rn					Rd				
		U															op														

Vector

```
UADALP <Vd>.<Ta>, <Vn>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDL, UADDL2

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UADDL` instruction extracts each source vector from the lower half of each source register, while the `UADDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	0	0	0	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
UADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	0	0	1	0	1	0	Rn					Rd				
		U															op														

Vector

```
UADDLP <Vd>.<Ta>, <Vn>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDLV

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	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	1	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
		U																													

Advanced SIMD

```
UADDLV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	H
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d] = sum<2*esize-1:0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDW, UADDW2

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The `UADDW` instruction extracts vector elements from the lower half of the second source register, while the `UADDW2` instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	0	0	1	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
UADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UBFIZ

Unsigned Bitfield Insert in Zero zeroes the destination register and copies any number of contiguous bits from a source register into any position in the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1	0	1	0	0	1	1	0	N	immr	imms	Rn	Rd
	opc												

32-bit (sf == 0 && N == 0)

```
UBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit (sf == 1 && N == 1)

```
UBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UBFM

Unsigned Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, with zeros in the upper and lower bits.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1	0	1	0	0	1	1	0	N	immr	imms	Rn	Rd
	opc												

32-bit (sf == 0 && N == 0)

```
UBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit (sf == 1 && N == 1)

```
UBFM <Xd>, <Xn>, #<immr>, #<imms>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer R;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then ReservedValue();
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then ReservedValue();

R = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
<imms>	For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Of variant	Is preferred when
LSL (immediate)	32-bit	`imms != '011111' && imms + 1 == immr`
LSL (immediate)	64-bit	`imms != '111111' && imms + 1 == immr`
LSR (immediate)	32-bit	`imms == '011111'`
LSR (immediate)	64-bit	`imms == '111111'`
UBFIZ		`UInt(imms) < UInt(immr)`
UBFX		`BFXPreferred(sf, opc<1>, imms, immr)`
UXTB		`immr == '000000' && imms == '000111'`

Alias	Of variant	Is preferred when
UXTH		`immr == '000000' && imms == '001111'`

Operation

```
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, R) AND wmask;

// combine extension bits and result bits
X[d] = bot AND tmask;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UBFX

Unsigned Bitfield Extract extracts any number of adjacent bits at any position from a register, zero-extends them to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
sf	1	0	1	0	0	1	1	0	N	immr	imms	Rn	Rd
	opc												

32-bit (sf == 0 && N == 0)

```
UBFX <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

64-bit (sf == 1 && N == 1)

```
UBFX <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UCVTF (vector, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn					Rd				
		U							immh																						

Scalar

```
UCVTF <V><d>, <V><n>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn					Rd				
		U							immh																						

Vector

```
UCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then ReservedValue();
if immh<3>:Q == '10' then ReservedValue();
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<V>	Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UCVTF (vector, integer)

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U																													

Scalar half precision

```
UCVTF <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U																													

Scalar single-precision and double-precision

```
UCVTF <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U																													

Vector half precision

```
UCVTF <Vd>.<T>, <Vn>.<T>
```

```
if !HaveFP16Ext() then UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				
		U																													

Vector single-precision and double-precision

```
UCVTF <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then ReservedValue();
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector half precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
FPRounding rounding = FPRoundingMode(FPCR);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		0	0	0	0	1	1	scale						Rn					Rd				
											rmode		opcode																		

32-bit to half-precision (sf == 0 && type == 11)
(ARMv8.2)

```
UCVTF <Hd>, <Wn>, #<fbits>
```

32-bit to single-precision (sf == 0 && type == 00)

```
UCVTF <Sd>, <Wn>, #<fbits>
```

32-bit to double-precision (sf == 0 && type == 01)

```
UCVTF <Dd>, <Wn>, #<fbits>
```

64-bit to half-precision (sf == 1 && type == 11)
(ARMv8.2)

```
UCVTF <Hd>, <Xn>, #<fbits>
```

64-bit to single-precision (sf == 1 && type == 00)

```
UCVTF <Sd>, <Xn>, #<fbits>
```

64-bit to double-precision (sf == 1 && type == 01)

```
UCVTF <Dd>, <Xn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

if sf == '0' && scale<5> == '0' then UnallocatedEncoding();
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale". For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = FixedToFP(intval, fracbits, TRUE, FPCR, rounding);
V[d] = fltval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	type		1	0	0	0	1	1	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																		

32-bit to half-precision (sf == 0 && type == 11)
(ARMv8.2)

```
UCVTF <Hd>, <Wn>
```

32-bit to single-precision (sf == 0 && type == 00)

```
UCVTF <Sd>, <Wn>
```

32-bit to double-precision (sf == 0 && type == 01)

```
UCVTF <Dd>, <Wn>
```

64-bit to half-precision (sf == 1 && type == 11)
(ARMv8.2)

```
UCVTF <Hd>, <Xn>
```

64-bit to single-precision (sf == 1 && type == 00)

```
UCVTF <Sd>, <Xn>
```

64-bit to double-precision (sf == 1 && type == 01)

```
UCVTF <Dd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case type of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UnallocatedEncoding();
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UnallocatedEncoding();

rounding = FPRoundingMode(FPCR);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = FixedToFP(intval, 0, TRUE, FPCR, rounding);
V[d] = fltval;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	0	0	1	0	Rn					Rd				
																					o1										

32-bit (sf == 0)

```
UDIV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
UDIV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(Int(operand2, TRUE)));

X[d] = result<datasize-1:0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDOT (by element)

Dot Product unsigned arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

From ARMv8.2, this is an OPTIONAL instruction.

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				1	1	1	0	H	0	Rn					Rd				
		U																													

Vector

```
UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then ReservedValue();
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the element index, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDOT (vector)

Dot Product unsigned arithmetic (vector). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

From ARMv8.2, this is an OPTIONAL instruction.

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Three registers of the same type

(ARMv8.2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		0	Rm					1	0	0	1	0	1	Rn					Rd				
		U																													

Three registers of the same type

```
UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then ReservedValue();
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHADD

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see *URHADD*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	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	1	Rn					Rd				
		U																													

Three registers of the same type

```
UHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHSUB

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	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	1	Rn					Rd				
		U																													

Three registers of the same type

```
UHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMULL.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	1	0	1	Rm					0	Ra					Rn					Rd				
								U								o0															

64-bit

```
UMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
UMULL	`Ra == '11111'`

Operation

```
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, TRUE) + (Int(operand1, TRUE) * Int(operand2, TRUE));

X[d] = result<63:0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAX

Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	1	1	0	0	1	Rn					Rd				
		U																		o1											

Three registers of the same type

```
UMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAXP

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	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	1	Rn					Rd				
		U																		o1											

Three registers of the same type

```
UMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAXV

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	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	1	0	0	0	0	1	0	1	0	1	0	Rn					Rd				
		U													op																

Advanced SIMD

```
UMAXV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMIN

Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	1	1	0	1	1	Rn					Rd				
		U																		o1											

Three registers of the same type

```
UMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMINP

Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	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	1	1	Rn					Rd				
		U																		o1											

Three registers of the same type

```
UMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMINV

Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	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	1	0	0	0	1	1	0	1	0	1	0	Rn					Rd				
		U													op																

Advanced SIMD

```
UMINV <V><d>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then ReservedValue();
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLAL, UMLAL2 (by element)

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLAL` instruction extracts vector elements from the lower half of the first source register, while the `UMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				0	0	1	0	H	0	Rn					Rd				
		U															o2														

Vector

```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLAL` instruction extracts vector elements from the lower half of the first source register, while the `UMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					1	0	0	0	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLSL` instruction extracts vector elements from the lower half of the first source register, while the `UMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				0	1	1	0	H	0	Rn					Rd				
		U															o2														

Vector

```
UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMLSL` instruction extracts each source vector from the lower half of each source register, while the `UMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					1	0	1	0	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMNEGL

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of UMSUBL. This means:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	1	0	1	Rm					1	1	1	1	1	1	Rn					Rd				
								U								o0			Ra												

64-bit

```
UMNEGL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMSUBL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMSUBL gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMOV

Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (to general).

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	0	0	1	1	1	0	0	0	0	imm5	0	0	1	1	1	1	Rn	Rd

32-bit (Q == 0)

```
UMOV <Wd>, <Vn>.<Ts>[<index>]
```

64-bit (Q == 1 && imm5 == x1000)

```
UMOV <Xd>, <Vn>.<Ts>[<index>]
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when '0xxxx1' size = 0;     // UMOV Wd, Vn.B
    when '0xxx10' size = 1;     // UMOV Wd, Vn.H
    when '0xx100' size = 2;     // UMOV Wd, Vn.S
    when '1x1000' size = 3;     // UMOV Xd, Vn.D
    otherwise UnallocatedEncoding();

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in “imm5”:

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

For the 64-bit variant: is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	RESERVED
xxx10	RESERVED
xx100	RESERVED
x1000	D

<index> For the 32-bit variant: is the element index encoded in “imm5”:

imm5	<index>
xx000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>

For the 64-bit variant: is the element index encoded in "imm5<4>".

Alias Conditions

Alias	Is preferred when
MOV (to general)	`imm5 == 'x1000'`
MOV (to general)	`imm5 == 'xx100'`

Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];

X[d] = ZeroExtend(Elem[operand, index, esize], datasize);
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMSUBL

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMNEGL.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	1	0	1	Rm					1	Ra					Rn					Rd				
								U								o0															

64-bit

```
UMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
UMNEGL	`Ra == '11111'`

Operation

```
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, TRUE) - (Int(operand1, TRUE) * Int(operand2, TRUE));
X[d] = result<63:0>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	1	1	0	Rm					0	(1)	(1)	(1)	(1)	(1)	Rn					Rd				
								U									Ra														

64-bit

```
UMULH <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
bits(64) operand1 = X[n];
bits(64) operand2 = X[m];

integer result;

result = Int(operand1, TRUE) * Int(operand2, TRUE);

X[d] = result<127:64>;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMULL` instruction extracts vector elements from the lower half of the first source register, while the `UMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				1	0	1	0	H	0	Rn					Rd				
		U																													

Vector

```
UMULL{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 UnallocatedEncoding();

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULL, UMULL2 (vector)

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMULL` instruction extracts each source vector from the lower half of each source register, while the `UMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23 22	21	20 19 18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	1	0	1	1	1	0	size	1	Rm	1	1	0	0	0	0	Rn	Rd
		U																

Three registers, not all the same type

```
UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULL

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of UMADDL. This means:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	1	0	1	Rm					0	1	1	1	1	1	Rn					Rd				
								U								o0			Ra												

64-bit

```
UMULL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMADDL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMADDL gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQADD

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	Rm					0	0	0	0	1	1	Rn					Rd				
		U																													

Scalar

```
UQADD <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	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	1	1	Rn					Rd				
		U																													

Vector

```
UQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQRSHL

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see *UQSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	Rm					0	1	0	1	1	1	Rn					Rd				
		U																	R	S											

Scalar

```
UQRSHL <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	1	0	1	1	1	Rn					Rd				
		U																	R	S											

Vector

```
UQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQRSHRN, UQRSHRN2

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *UQSHRN*.

The `UQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	0	0	1	1	1	Rn					Rd				
		U							immh											op											

Scalar

```
UQRSHRN <Vb><d>, <Va><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	1	1	Rn					Rd				
		U							immh											op											

Vector

```
UQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSHL (immediate)

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see *UQRSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	1	1	1	0	1	Rn					Rd				
		U							immh										op												

Scalar

```
UQSHL <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	1	1	0	1	Rn					Rd				
		U							immh										op												

Vector

```
UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UnallocatedEncoding();
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSHL (register)

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see *UQRSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	Rm					0	1	0	0	1	1	Rn					Rd				
		U																	R	S											

Scalar

```
UQSHL <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	1	0	0	1	1	Rn					Rd				
		U																	R	S											

Vector

```
UQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSHRN, UQSHRN2

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *UQRSHRN*.

The `UQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn					Rd				
		U							immh											op											

Scalar

```
UQSHRN <Vb><d>, <Va><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then ReservedValue();
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn					Rd				
		U							immh											op											

Vector

```
UQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSUB

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set. Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	Rm					0	0	1	0	1	1	Rn					Rd				
		U																													

Scalar

```
UQSUB <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	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	1	1	Rn					Rd				
		U																													

Vector

```
UQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQXTN, UQXTN2

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `UQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	0	0	0	0	1	0	1	0	0	1	0	Rn					Rd				
		U																													

Scalar

```
UQXTN <Vb><d>, <Va><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	1	0	1	0	0	1	0	Rn					Rd				
		U																													

Vector

```
UQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URECPE

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn					Rd				

Vector

```
URECPE <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then ReservedValue();
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRecipEstimate(element);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URHADD

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see *UHADD*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	0	0	1	0	1	Rn					Rd				
		U																													

Three registers of the same type

```
URHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1+element2+1)<esize:1>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URSHL

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	Rm					0	1	0	1	0	1	Rn					Rd				
		U																	R	S											

Scalar

```
URSHL <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	1	0	1	0	1	Rn					Rd				
		U																	R	S											

Vector

```
URSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URSHR

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *USHR*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	0	1	0	0	1	Rn					Rd				
		U							immh									o1	o0												

Scalar

```
URSHR <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	0	1	0	0	1	Rn					Rd				
		U							immh									o1	o0												

Vector

```
URSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V>	Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URSQRTE

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn					Rd				

Vector

```
URSQRTE <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then ReservedValue();
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRSqrtEstimate(element);

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URSRA

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *USRA*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	0	1	1	0	1	Rn					Rd				
		U							immh									o1	o0												

Scalar

```
URSRA <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	0	1	1	0	1	Rn					Rd				
		U							immh									o1	o0												

Vector

```
URSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USHL

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see *URSHL*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	Rm					0	1	0	0	0	1	Rn					Rd				
		U																	R	S											

Scalar

```
USHL <V><d>, <V><n>, <V><m>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then ReservedValue();
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	1	0	0	0	1	Rn					Rd				
		U																	R	S											

Vector

```
USHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USHLL, USHLL2

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `USHLL` instruction extracts vector elements from the lower half of the source register, while the `USHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias UXTL, UXTL2.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	1	0	0	1	Rn					Rd				
		U							immh																						

Vector

```
USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	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;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USHR

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *URSHR*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	0	0	0	0	1	Rn					Rd				
		U							immh									o1	o0												

Scalar

```
USHR <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	0	0	0	0	1	Rn					Rd				
		U							immh									o1	o0												

Vector

```
USHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V>	Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USQADD

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	0	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
		U																													

Scalar

```
USQADD <V><d>, <V><n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	0	0	0	1	1	1	0	Rn					Rd				
		U																													

Vector

```
USQADD <Vd>.<T>, <Vn>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USRA

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *URSRA*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	0	0	1	0	1	Rn					Rd				
		U							immh									o1	o0												

Scalar

```
USRA <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then ReservedValue();
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	0	0	1	0	1	Rn					Rd				
		U							immh									o1	o0												

Vector

```
USRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then ReservedValue();
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USUBL, USUBL2

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The `USUBL` instruction extracts each source vector from the lower half of each source register, while the `USUBL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	0	1	0	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
USUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USUBW, USUBW2

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The USUBW instruction extracts vector elements from the lower half of the first source register, while the USUBW2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	Rm					0	0	1	1	0	0	Rn					Rd				
		U																o1													

Three registers, not all the same type

```
USUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	Rn					Rd				
sf	opc								N			immr						imms													

32-bit

```
UXTB <Wd>, <Wn>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	Rn					Rd				
sf	opc								N			immr						imms													

32-bit

```
UXTH <Wd>, <Wn>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																		

Vector

```
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.

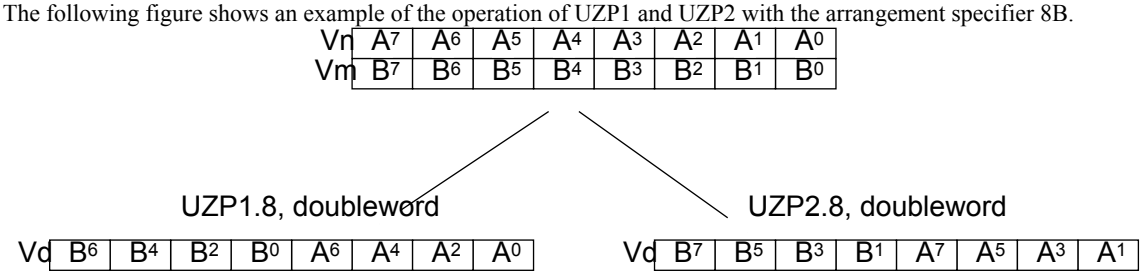
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UZP1

Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can be used with `UZP2` to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		0	Rm					0	0	0	1	1	0	Rn					Rd				
																	op														

Advanced SIMD

```
UZP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;
integer e;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

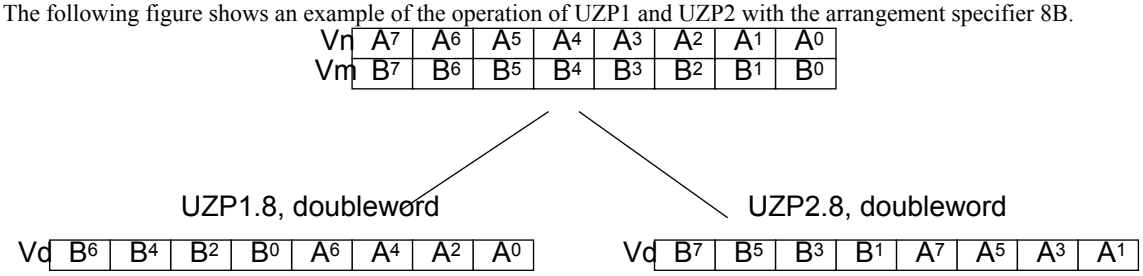
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UZP2

Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can be used with `UZP1` to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		0	Rm					0	1	0	1	1	0	Rn					Rd				
																	op														

Advanced SIMD

```
UZP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;
integer e;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event*.

As described in *Wait For Event mechanism and Send event*, the execution of a `WFE` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions*.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions*.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0	1	1	1	1	1
																					CRm				op2						

System

```
WFE
```

```
// Empty.
```

Operation

```
if IsEventRegisterSet() then
    ClearEventRegister();
else
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, TRUE);
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, TRUE);
    if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, TRUE);
    WaitForEvent();
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see *Wait For Interrupt*.

As described in *Wait For Interrupt*, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions*.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions*.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1
																					CRm				op2						

System

```
WFI
```

```
// Empty.
```

Operation

```
if !InterruptPending() then
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, FALSE);
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, FALSE);
    if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, FALSE);
    WaitForInterrupt();
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

XAR

Exclusive OR and Rotate performs a bitwise exclusive OR of the 128-bit vectors in the two source SIMD&FP registers, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when *ARMv8.2-SHA* is implemented.

Advanced SIMD

(ARMv8.2)

31	30	29	28	27	26	25	24	23	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	0	0	Rm	imm6	Rn	Rd

Advanced SIMD

```
XAR <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>
```

```
if !HaveCryptoExt2() then UnallocatedEncoding();
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<imm6>	Is a rotation right, encoded in "imm6".

Operation

```
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) tmp;
tmp = Vn EOR Vm;
V[d] = ROR(tmp<127:64>, UInt(imm6)):ROR(tmp<63:0>, UInt(imm6));
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

XPACD, XPACI, XPACLRI

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for `XPACI` and `XPACD`, and is in LR for `XPACLRI`.

The `XPACD` instruction is used for data addresses, and `XPACI` and `XPACLRI` are used for instruction addresses.

It has encodings from 2 classes: Integer and System

Integer

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	1	0	0	0	D	1	1	1	1	1	Rd				
																								Rn							

XPACD (D == 1)

```
XPACD <Xd>
```

XPACI (D == 0)

```
XPACI <Xd>
```

```
boolean data = (D == '1');
integer d = UInt(Rd);

if !HavePACExt() then
    UnallocatedEncoding();
```

System

(ARMv8.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1

System

```
XPACLRI
```

```
integer d = 30;
boolean data = FALSE;
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

```
if HavePACExt() then
    X[d] = Strip(X[d], data);
```


Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

XTN, XTN2

Extract Narrow. This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

The `XTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `XTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		1	0	0	0	0	1	0	0	1	0	1	0	Rn					Rd				

Vector

```
XTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see *The YIELD instruction*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1
																					CRm				op2						

System

```
YIELD
```

```
// Empty.
```

Operation

```
Hint_Yield();
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

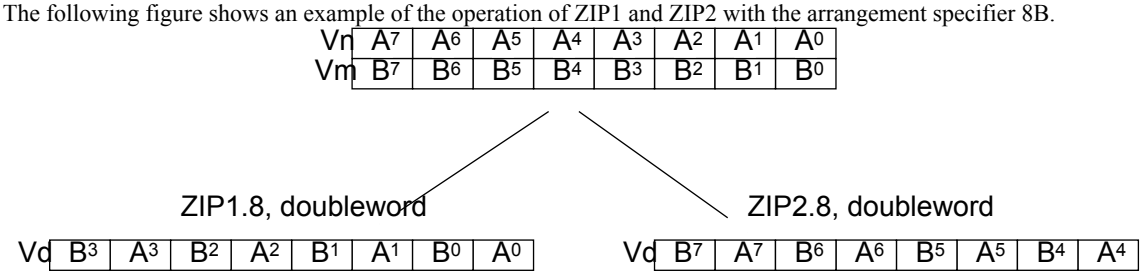
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ZIP1

Zip vectors (primary). This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

This instruction can be used with `ZIP2` to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		0	Rm					0	0	1	1	1	0	Rn					Rd				
																	op														

Advanced SIMD

```
ZIP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;
integer p;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

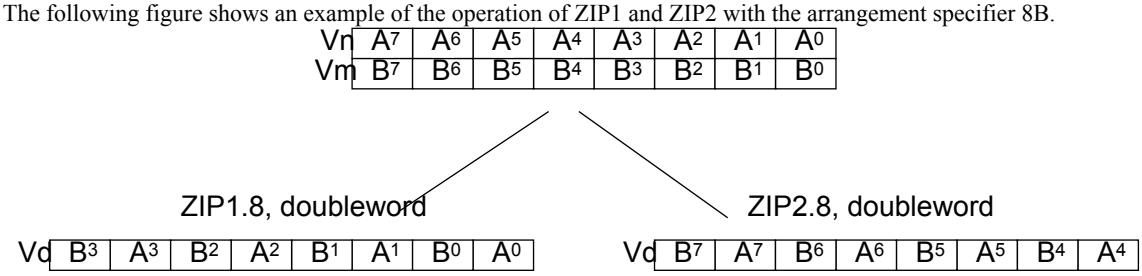
Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ZIP2

Zip vectors (secondary). This instruction reads adjacent vector elements from the lower half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

This instruction can be used with `ZIP1` to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		0	Rm					0	1	1	1	1	0	Rn					Rd				
																	op														

Advanced SIMD

```
ZIP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then ReservedValue();
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;
integer p;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

Top-level encodings for A64

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			op0																												

Decode fields op0	Instruction details
00xx	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

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 op0	Instruction details
00x	PC-rel. addressing
01x	Add/subtract (immediate)
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 op	Instruction Details
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	shift		imm12														Rn					Rd		

Decode fields sf	op	S	shift	Instruction Details
			1x	UNALLOCATED
0	0	0		ADD (immediate) — 32-bit
0	0	1		ADDS (immediate) — 32-bit

Decode fields				Instruction Details
sf	op	S	shift	
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

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		MOVN — 32-bit
0	10		MOVZ — 32-bit
0	11		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 sf	opc	N	Instruction Details
	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 sf	op21	N	o0	imms	Instruction Details
	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																									

Decode fields op0	op1	Instruction details
010	0xxx	Conditional branch (immediate)
010	1xxx	UNALLOCATED
110	00xx	Exception generation
110	0100	System
110	0101	UNALLOCATED
110	011x	UNALLOCATED
110	1xxx	Unconditional branch (register)
x00		Unconditional branch (immediate)
x01	0xxx	Compare and branch (immediate)
x01	1xxx	Test and branch (immediate)
x11		UNALLOCATED

Conditional branch (immediate)

These instructions are under Branches, Exception Generating and System instructions.

31	30	29	28	27	26	25	24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
0	1	0	1	0	1	0	o1	imm19	o0	cond

Decode fields o1	o0	Instruction Details
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 opc	op2	LL	Instruction Details
	xx1		UNALLOCATED
	x1x		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		UNALLOCATED
100	000		UNALLOCATED
101	000	00	UNALLOCATED
101	000	01	DCPS1
101	000	10	DCPS2
101	000	11	DCPS3
11x	000		UNALLOCATED

System

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	op0	op1	CRn	CRm	op2	Rt

Decode fields L	op0	op1	CRn	CRm	op2	Rt	Instruction Details	Architecture Version
0	00		000x				UNALLOCATED	-
0	00		0100			!= 11111	UNALLOCATED	-

L	op0	op1	CRn	CRm	op2	Rt	Instruction Details	Architecture Version
			Decode fields					
0	00		0100			11111	MSR (immediate)	-
0	00		0101				UNALLOCATED	-
0	00		011x				UNALLOCATED	-
0	00		1xxx				UNALLOCATED	-
0	00	xx0	001x				UNALLOCATED	-
0	00	x0x	001x				UNALLOCATED	-
0	00	011	001x			!= 11111	UNALLOCATED	-
0	00	011	0010	!= 00x0		11111	HINT — hints 8 to 15, and 24 to 127	-
0	00	011	0010	0000	000	11111	NOP	-
0	00	011	0010	0000	001	11111	YIELD	-
0	00	011	0010	0000	010	11111	WFE	-
0	00	011	0010	0000	011	11111	WFI	-
0	00	011	0010	0000	100	11111	SEV	-
0	00	011	0010	0000	101	11111	SEVL	-
0	00	011	0010	0000	11x	11111	HINT — hints 6 and 7	-
0	00	011	0010	0000	111	11111	XPACD, XPACI, XPACLRI	ARMv8.3
0	00	011	0010	0001	000	11111	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA1716	ARMv8.3
0	00	011	0010	0001	010	11111	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB1716	ARMv8.3
0	00	011	0010	0001	100	11111	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA1716	ARMv8.3
0	00	011	0010	0001	110	11111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB1716	ARMv8.3
0	00	011	0010	0010	!= 00x	11111	HINT — hints 17 to 23	-
0	00	011	0010	0010	000	11111	ESB	ARMv8.2
0	00	011	0010	0010	001	11111	PSB CSYNC	ARMv8.2
0	00	011	0010	0011	000	11111	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIAZ	ARMv8.3
0	00	011	0010	0011	001	11111	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIASP	ARMv8.3
0	00	011	0010	0011	010	11111	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBZ	ARMv8.3
0	00	011	0010	0011	011	11111	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBSP	ARMv8.3
0	00	011	0010	0011	100	11111	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIAZ	ARMv8.3
0	00	011	0010	0011	101	11111	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIASP	ARMv8.3
0	00	011	0010	0011	110	11111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBZ	ARMv8.3
0	00	011	0010	0011	111	11111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBSP	ARMv8.3
0	00	011	0011		000		UNALLOCATED	-
0	00	011	0011		001		UNALLOCATED	-
0	00	011	0011		010	11111	CLREX	-
0	00	011	0011		011		UNALLOCATED	-
0	00	011	0011		100	11111	DSB	-
0	00	011	0011		101	11111	DMB	-

L	op0	op1	CRn	CRm	op2	Rt	Instruction Details	Architecture Version
		Decode fields						
0	00	011	0011		110	11111	ISB	-
0	00	011	0011		111		UNALLOCATED	-
0	00	1xx	001x				UNALLOCATED	-
0	01						SYS	-
0	1x						MSR (register)	-
1	00						UNALLOCATED	-
1	01						SYSL	-
1	1x						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	op3	Rn	op4	Instruction Details	Architecture Version
		Decode fields				
				!= 00000	UNALLOCATED	-
		!= 000000			UNALLOCATED	-
	!= 11111				UNALLOCATED	-
0000	11111	000000		00000	BR	-
0000	11111	000010		11111	BRAA, BRAAZ, BRAB, BRABZ — key A, zero modifier	ARMv8.3
0000	11111	000011		11111	BRAA, BRAAZ, BRAB, BRABZ — key B, zero modifier	ARMv8.3
0001	11111	000000		00000	BLR	-
0001	11111	000010		11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, zero modifier	ARMv8.3
0001	11111	000011		11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, zero modifier	ARMv8.3
0010	11111	000000		00000	RET	-
0010	11111	000010	11111	11111	RETAA, RETAB — RETAA	ARMv8.3
0010	11111	000011	11111	11111	RETAA, RETAB — RETAB	ARMv8.3
0011					UNALLOCATED	-
010x			!= 11111		UNALLOCATED	-
0100	11111	000000	11111	00000	ERET	-
0100	11111	000010	11111	11111	ERETAA, ERETAB — ERETAA	ARMv8.3
0100	11111	000011	11111	11111	ERETAA, ERETAB — ERETAB	ARMv8.3
0101	11111	000000	11111	00000	DRPS	-
011x					UNALLOCATED	-
1xxx					UNALLOCATED	-
1000	11111	000010			BRAA, BRAAZ, BRAB, BRABZ — key A, register modifier	ARMv8.3
1000	11111	000011			BRAA, BRAAZ, BRAB, BRABZ — key B, register modifier	ARMv8.3
1001	11111	000010			BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, register modifier	ARMv8.3

Decode fields opc	op2	op3	Rn	op4	Instruction Details	Architecture Version
1001	11111	000011			BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, register modifier	ARMv8.3

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 op	Instruction Details
0	B
1	BL

Compare and branch (immediate)

These instructions are under Branches, Exception Generating and System instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	0	1	0	op	imm19																			Rt				

Decode fields sf	op	Instruction Details
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	0	1	1	0	1	1	op	b40					imm14														Rt				

Decode fields op	Instruction Details
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		op1		1	op2	0	op3		op4											op5											

Decode fields op0	op1	op2	op3	op4	op5	Instruction details
0	00	1	00	000000		Advanced SIMD load/store multiple structures
0	00	1	01	0xxxxx		Advanced SIMD load/store multiple structures (post-indexed)
0	00	1	0x	1xxxxx		UNALLOCATED
0	00	1	10	x00000		Advanced SIMD load/store single structure

0	00	1	11			Advanced SIMD load/store single structure (post-indexed)
0	00	1	x0	x1xxxx		UNALLOCATED
0	00	1	x0	xx1xxx		UNALLOCATED
0	00	1	x0	xxx1xx		UNALLOCATED
0	00	1	x0	xxxx1x		UNALLOCATED
0	00	1	x0	xxxxx1		UNALLOCATED
1	00	1				UNALLOCATED
	00	0	0x			Load/store exclusive
	00	0	1x			UNALLOCATED
	01		0x			Load register (literal)
	01		1x			UNALLOCATED
	10		00			Load/store no-allocate pair (offset)
	10		01			Load/store register pair (post-indexed)
	10		10			Load/store register pair (offset)
	10		11			Load/store register pair (pre-indexed)
	11		0x	0xxxxx	00	Load/store register (unscaled immediate)
	11		0x	0xxxxx	01	Load/store register (immediate post-indexed)
	11		0x	0xxxxx	10	Load/store register (unprivileged)
	11		0x	0xxxxx	11	Load/store register (immediate pre-indexed)
	11		0x	1xxxxx	00	Atomic memory operations
	11		0x	1xxxxx	10	Load/store register (register offset)
	11		0x	1xxxxx	x1	Load/store register (pac)
	11		1x			Load/store register (unsigned immediate)

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 L	opcode	Instruction Details
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
1	0010	LD1 (multiple structures) — four registers
1	0011	UNALLOCATED

Decode fields L	opcode	Instruction Details
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 L	Rm	opcode	Instruction Details
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
1	!= 11111	0100	LD3 (multiple structures) — register offset
1	!= 11111	0110	LD1 (multiple structures) — three registers, register offset

L	Decode fields Rm	opcode	Instruction Details
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		Rn						Rt			

L	R	Decode fields opcode	S	size	Instruction Details
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
0	1	101		00	ST4 (single structure) — 32-bit
0	1	101		10	UNALLOCATED

Decode fields					Instruction Details
L	R	opcode	S	size	
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				

L	R	Rm	opcode	S	size	Instruction Details
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

Decode fields						Instruction Details
L	R	Rm	opcode	S	size	
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 exclusive

These instructions are under Loads and Stores.

31 30	29 28 27 26 25 24	23	22	21	20 19 18 17 16	15	14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
size	0 0 1 0 0 0	o2	L	o1	Rs	o0	Rt2	Rn	Rt

Decode fields						Instruction Details	Architecture Version
size	o2	L	o1	o0	Rt2		
	1		1		!= 11111	UNALLOCATED	-
0x	0		1		!= 11111	UNALLOCATED	-
00	0	0	0	0		STXRB	-
00	0	0	0	1		STLXRB	-
00	0	0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit, no memory ordering	ARMv8.1
00	0	0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit, release	ARMv8.1
00	0	1	0	0		LDXRB	-
00	0	1	0	1		LDAXRB	-
00	0	1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit, acquire	ARMv8.1
00	0	1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit, acquire and release	ARMv8.1
00	1	0	0	0		STLLRB	ARMv8.1
00	1	0	0	1		STLRB	-
00	1	0	1	0	11111	CASB, CASAB, CASALB, CASLB — no memory ordering	ARMv8.1
00	1	0	1	1	11111	CASB, CASAB, CASALB, CASLB — release	ARMv8.1
00	1	1	0	0		LDLARB	ARMv8.1
00	1	1	0	1		LDARB	-
00	1	1	1	0	11111	CASB, CASAB, CASALB, CASLB — acquire	ARMv8.1
00	1	1	1	1	11111	CASB, CASAB, CASALB, CASLB — acquire and release	ARMv8.1
01	0	0	0	0		STXRH	-
01	0	0	0	1		STLXRH	-
01	0	0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit, no memory ordering	ARMv8.1

Decode fields						Instruction Details	Architecture Version
size	o2	L	o1	o0	Rt2		
01	0	0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit, release	ARMv8.1
01	0	1	0	0		LDXRH	-
01	0	1	0	1		LDAXRH	-
01	0	1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit, acquire	ARMv8.1
01	0	1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit, acquire and release	ARMv8.1
01	1	0	0	0		STLLRH	ARMv8.1
01	1	0	0	1		STLRH	-
01	1	0	1	0	11111	CASH, CASAH, CASALH, CASLH — no memory ordering	ARMv8.1
01	1	0	1	1	11111	CASH, CASAH, CASALH, CASLH — release	ARMv8.1
01	1	1	0	0		LDLARH	ARMv8.1
01	1	1	0	1		LDARH	-
01	1	1	1	0	11111	CASH, CASAH, CASALH, CASLH — acquire	ARMv8.1
01	1	1	1	1	11111	CASH, CASAH, CASALH, CASLH — acquire and release	ARMv8.1
10	0	0	0	0		STXR — 32-bit	-
10	0	0	0	1		STLXR — 32-bit	-
10	0	0	1	0		STXP — 32-bit	-
10	0	0	1	1		STLXP — 32-bit	-
10	0	1	0	0		LDXR — 32-bit	-
10	0	1	0	1		LDAXR — 32-bit	-
10	0	1	1	0		LDXP — 32-bit	-
10	0	1	1	1		LDAXP — 32-bit	-
10	1	0	0	0		STLLR — 32-bit	ARMv8.1
10	1	0	0	1		STLR — 32-bit	-
10	1	0	1	0	11111	CAS, CASA, CASAL, CASL — 32-bit, no memory ordering	ARMv8.1
10	1	0	1	1	11111	CAS, CASA, CASAL, CASL — 32-bit, release	ARMv8.1
10	1	1	0	0		LDLAR — 32-bit	ARMv8.1
10	1	1	0	1		LDAR — 32-bit	-
10	1	1	1	0	11111	CAS, CASA, CASAL, CASL — 32-bit, acquire	ARMv8.1
10	1	1	1	1	11111	CAS, CASA, CASAL, CASL — 32-bit, acquire and release	ARMv8.1
11	0	0	0	0		STXR — 64-bit	-
11	0	0	0	1		STLXR — 64-bit	-
11	0	0	1	0		STXP — 64-bit	-
11	0	0	1	1		STLXP — 64-bit	-
11	0	1	0	0		LDXR — 64-bit	-
11	0	1	0	1		LDAXR — 64-bit	-
11	0	1	1	0		LDXP — 64-bit	-
11	0	1	1	1		LDAXP — 64-bit	-
11	1	0	0	0		STLLR — 64-bit	ARMv8.1
11	1	0	0	1		STLR — 64-bit	-
11	1	0	1	0	11111	CAS, CASA, CASAL, CASL — 64-bit, no memory ordering	ARMv8.1
11	1	0	1	1	11111	CAS, CASA, CASAL, CASL — 64-bit, release	ARMv8.1
11	1	1	0	0		LDLAR — 64-bit	ARMv8.1
11	1	1	0	1		LDAR — 64-bit	-
11	1	1	1	0	11111	CAS, CASA, CASAL, CASL — 64-bit, acquire	ARMv8.1
11	1	1	1	1	11111	CAS, CASA, CASAL, CASL — 64-bit, acquire and release	ARMv8.1

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
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 opc	V	L	Instruction Details
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	UNALLOCATED
01	0	1	LDPSW

Decode fields opc	V	L	Instruction Details
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 opc	V	L	Instruction Details
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	UNALLOCATED
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 opc	V	L	Instruction Details
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	UNALLOCATED
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

Decode fields opc	V	L	Instruction Details
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 size	V	opc	Instruction Details
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
10	1	01	LDUR (SIMD&FP) — 32-bit
11	0	00	STUR — 64-bit
11	0	01	LDUR — 64-bit
11	0	10	PRFM (unscaled offset)
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 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
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

Decode fields size	V	opc	Instruction Details
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
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

size	V	A	R	o3	opc	Rt	Instruction Details	Architecture Version
	0			1	001		UNALLOCATED	-
	0			1	01x		UNALLOCATED	-
	0			1	101		UNALLOCATED	-
	0			1	11x		UNALLOCATED	-
	0	0		1	100		UNALLOCATED	-
	0	1	1	1	100		UNALLOCATED	-
	1						UNALLOCATED	-
00	0	0	0	0	000	!= 11111	LDADDB, LDADDAB, LDADDALB, LDADDLB — no memory ordering	ARMv8.1
00	0	0	0	0	000	11111	STADDB, STADDLB — no memory ordering	ARMv8.1
00	0	0	0	0	001	!= 11111	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — no memory ordering	ARMv8.1
00	0	0	0	0	001	11111	STCLRB, STCLRLB — no memory ordering	ARMv8.1
00	0	0	0	0	010	!= 11111	LDEORB, LDEORAB, LDEORALB, LDEORLB — no memory ordering	ARMv8.1
00	0	0	0	0	010	11111	STEORB, STEORLB — no memory ordering	ARMv8.1
00	0	0	0	0	011	!= 11111	LDSETB, LDSETAB, LDSETALB, LDSETLB — no memory ordering	ARMv8.1
00	0	0	0	0	011	11111	STSETB, STSETLB — no memory ordering	ARMv8.1
00	0	0	0	0	100	!= 11111	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — no memory ordering	ARMv8.1
00	0	0	0	0	100	11111	STSMAXB, STSMAXLB — no memory ordering	ARMv8.1
00	0	0	0	0	101	!= 11111	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — no memory ordering	ARMv8.1
00	0	0	0	0	101	11111	STSMINB, STSMINLB — no memory ordering	ARMv8.1
00	0	0	0	0	110	!= 11111	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — no memory ordering	ARMv8.1
00	0	0	0	0	110	11111	STUMAXB, STUMAXLB — no memory ordering	ARMv8.1
00	0	0	0	0	111	!= 11111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — no memory ordering	ARMv8.1
00	0	0	0	0	111	11111	STUMINB, STUMINLB — no memory ordering	ARMv8.1
00	0	0	0	1	000		SWPB, SWPAB, SWPALB, SWPLB — no memory ordering	ARMv8.1
00	0	0	1	0	000	!= 11111	LDADDB, LDADDAB, LDADDALB, LDADDLB — release	ARMv8.1
00	0	0	1	0	000	11111	STADDB, STADDLB — release	ARMv8.1
00	0	0	1	0	001	!= 11111	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — release	ARMv8.1
00	0	0	1	0	001	11111	STCLRB, STCLRLB — release	ARMv8.1
00	0	0	1	0	010	!= 11111	LDEORB, LDEORAB, LDEORALB, LDEORLB — release	ARMv8.1
00	0	0	1	0	010	11111	STEORB, STEORLB — release	ARMv8.1
00	0	0	1	0	011	!= 11111	LDSETB, LDSETAB, LDSETALB, LDSETLB — release	ARMv8.1
00	0	0	1	0	011	11111	STSETB, STSETLB — release	ARMv8.1
00	0	0	1	0	100	!= 11111	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — release	ARMv8.1
00	0	0	1	0	100	11111	STSMAXB, STSMAXLB — release	ARMv8.1

Decode fields							Instruction Details	Architecture Version
size	V	A	R	o3	opc	Rt		
00	0	0	1	0	101	!= 11111	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — release	ARMv8.1
00	0	0	1	0	101	11111	STSMINB, STSMINLB — release	ARMv8.1
00	0	0	1	0	110	!= 11111	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — release	ARMv8.1
00	0	0	1	0	110	11111	STUMAXB, STUMAXLB — release	ARMv8.1
00	0	0	1	0	111	!= 11111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — release	ARMv8.1
00	0	0	1	0	111	11111	STUMINB, STUMINLB — release	ARMv8.1
00	0	0	1	1	000		SWPB, SWPAB, SWPALB, SWPLB — release	ARMv8.1
00	0	1	0	0	000		LDADDB, LDADDAB, LDADDALB, LDADDLB — acquire	ARMv8.1
00	0	1	0	0	001		LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — acquire	ARMv8.1
00	0	1	0	0	010		LDEORB, LDEORAB, LDEORALB, LDEORLB — acquire	ARMv8.1
00	0	1	0	0	011		LDSETB, LDSETAB, LDSETALB, LDSETLB — acquire	ARMv8.1
00	0	1	0	0	100		LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — acquire	ARMv8.1
00	0	1	0	0	101		LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — acquire	ARMv8.1
00	0	1	0	0	110		LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — acquire	ARMv8.1
00	0	1	0	0	111		LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — acquire	ARMv8.1
00	0	1	0	1	000		SWPB, SWPAB, SWPALB, SWPLB — acquire	ARMv8.1
00	0	1	0	1	100		LDAPRB	ARMv8.3
00	0	1	1	0	000		LDADDB, LDADDAB, LDADDALB, LDADDLB — acquire and release	ARMv8.1
00	0	1	1	0	001		LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — acquire and release	ARMv8.1
00	0	1	1	0	010		LDEORB, LDEORAB, LDEORALB, LDEORLB — acquire and release	ARMv8.1
00	0	1	1	0	011		LDSETB, LDSETAB, LDSETALB, LDSETLB — acquire and release	ARMv8.1
00	0	1	1	0	100		LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — acquire and release	ARMv8.1
00	0	1	1	0	101		LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — acquire and release	ARMv8.1
00	0	1	1	0	110		LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — acquire and release	ARMv8.1
00	0	1	1	0	111		LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — acquire and release	ARMv8.1
00	0	1	1	1	000		SWPB, SWPAB, SWPALB, SWPLB — acquire and release	ARMv8.1
01	0	0	0	0	000	!= 11111	LDADDH, LDADDAH, LDADDALH, LDADDLH — no memory ordering	ARMv8.1
01	0	0	0	0	000	11111	STADDH, STADDLH — no memory ordering	ARMv8.1
01	0	0	0	0	001	!= 11111	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — no memory ordering	ARMv8.1
01	0	0	0	0	001	11111	STCLRH, STCLRLH — no memory ordering	ARMv8.1
01	0	0	0	0	010	!= 11111	LDEORH, LDEORAH, LDEORALH, LDEORLH — no memory ordering	ARMv8.1
01	0	0	0	0	010	11111	STEORH, STEORLH — no memory ordering	ARMv8.1
01	0	0	0	0	011	!= 11111	LDSETH, LDSETAH, LDSETALH, LDSETLH — no memory ordering	ARMv8.1

Decode fields							Instruction Details	Architecture Version
size	V	A	R	o3	opc	Rt		
01	0	0	0	0	011	11111	STSETH, STSETLH — no memory ordering	ARMv8.1
01	0	0	0	0	100	!= 11111	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — no memory ordering	ARMv8.1
01	0	0	0	0	100	11111	STSMAXH, STSMAXLH — no memory ordering	ARMv8.1
01	0	0	0	0	101	!= 11111	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — no memory ordering	ARMv8.1
01	0	0	0	0	101	11111	STSMINH, STSMINLH — no memory ordering	ARMv8.1
01	0	0	0	0	110	!= 11111	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — no memory ordering	ARMv8.1
01	0	0	0	0	110	11111	STUMAXH, STUMAXLH — no memory ordering	ARMv8.1
01	0	0	0	0	111	!= 11111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — no memory ordering	ARMv8.1
01	0	0	0	0	111	11111	STUMINH, STUMINLH — no memory ordering	ARMv8.1
01	0	0	0	1	000		SWPH, SWPAH, SWPALH, SWPLH — no memory ordering	ARMv8.1
01	0	0	1	0	000	!= 11111	LDADDH, LDADDAH, LDADDALH, LDADDLH — release	ARMv8.1
01	0	0	1	0	000	11111	STADDH, STADDLH — release	ARMv8.1
01	0	0	1	0	001	!= 11111	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — release	ARMv8.1
01	0	0	1	0	001	11111	STCLRH, STCLRLH — release	ARMv8.1
01	0	0	1	0	010	!= 11111	LDEORH, LDEORAH, LDEORALH, LDEORLH — release	ARMv8.1
01	0	0	1	0	010	11111	STEORH, STEORLH — release	ARMv8.1
01	0	0	1	0	011	!= 11111	LDSETH, LDSETAH, LDSETALH, LDSETLH — release	ARMv8.1
01	0	0	1	0	011	11111	STSETH, STSETLH — release	ARMv8.1
01	0	0	1	0	100	!= 11111	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — release	ARMv8.1
01	0	0	1	0	100	11111	STSMAXH, STSMAXLH — release	ARMv8.1
01	0	0	1	0	101	!= 11111	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — release	ARMv8.1
01	0	0	1	0	101	11111	STSMINH, STSMINLH — release	ARMv8.1
01	0	0	1	0	110	!= 11111	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — release	ARMv8.1
01	0	0	1	0	110	11111	STUMAXH, STUMAXLH — release	ARMv8.1
01	0	0	1	0	111	!= 11111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — release	ARMv8.1
01	0	0	1	0	111	11111	STUMINH, STUMINLH — release	ARMv8.1
01	0	0	1	1	000		SWPH, SWPAH, SWPALH, SWPLH — release	ARMv8.1
01	0	1	0	0	000		LDADDH, LDADDAH, LDADDALH, LDADDLH — acquire	ARMv8.1
01	0	1	0	0	001		LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — acquire	ARMv8.1
01	0	1	0	0	010		LDEORH, LDEORAH, LDEORALH, LDEORLH — acquire	ARMv8.1
01	0	1	0	0	011		LDSETH, LDSETAH, LDSETALH, LDSETLH — acquire	ARMv8.1
01	0	1	0	0	100		LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — acquire	ARMv8.1
01	0	1	0	0	101		LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — acquire	ARMv8.1
01	0	1	0	0	110		LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — acquire	ARMv8.1

Decode fields							Instruction Details	Architecture Version
size	V	A	R	o3	opc	Rt		
01	0	1	0	0	111		LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — acquire	ARMv8.1
01	0	1	0	1	000		SWPH, SWPAH, SWPALH, SWPLH — acquire	ARMv8.1
01	0	1	0	1	100		LDAPRH	ARMv8.3
01	0	1	1	0	000		LDADDH, LDADDAH, LDADDALH, LDADDLH — acquire and release	ARMv8.1
01	0	1	1	0	001		LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — acquire and release	ARMv8.1
01	0	1	1	0	010		LDEORH, LDEORAH, LDEORALH, LDEORLH — acquire and release	ARMv8.1
01	0	1	1	0	011		LDSETH, LDSETAH, LDSETALH, LDSETLH — acquire and release	ARMv8.1
01	0	1	1	0	100		LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — acquire and release	ARMv8.1
01	0	1	1	0	101		LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — acquire and release	ARMv8.1
01	0	1	1	0	110		LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — acquire and release	ARMv8.1
01	0	1	1	0	111		LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — acquire and release	ARMv8.1
01	0	1	1	1	000		SWPH, SWPAH, SWPALH, SWPLH — acquire and release	ARMv8.1
10	0	0	0	0	000	!= 11111	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	000	11111	STADD, STADDL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	001	!= 11111	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	001	11111	STCLR, STCLRL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	010	!= 11111	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	010	11111	STEOR, STEORL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	011	!= 11111	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	011	11111	STSET, STSETL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	100	!= 11111	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	100	11111	STSMAX, STSMAXL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	101	!= 11111	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	101	11111	STSMIN, STSMINL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	110	!= 11111	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	110	11111	STUMAX, STUMAXL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	111	!= 11111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	0	111	11111	STUMIN, STUMINL — 32-bit, no memory ordering	ARMv8.1
10	0	0	0	1	000		SWP, SWPA, SWPAL, SWPL — 32-bit, no memory ordering	ARMv8.1
10	0	0	1	0	000	!= 11111	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit, release	ARMv8.1
10	0	0	1	0	000	11111	STADD, STADDL — 32-bit, release	ARMv8.1
10	0	0	1	0	001	!= 11111	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit, release	ARMv8.1

Decode fields							Instruction Details	Architecture Version
size	V	A	R	o3	opc	Rt		
10	0	0	1	0	001	11111	STCLR, STCLRL — 32-bit, release	ARMv8.1
10	0	0	1	0	010	!= 11111	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit, release	ARMv8.1
10	0	0	1	0	010	11111	STEOR, STEORL — 32-bit, release	ARMv8.1
10	0	0	1	0	011	!= 11111	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit, release	ARMv8.1
10	0	0	1	0	011	11111	STSET, STSETL — 32-bit, release	ARMv8.1
10	0	0	1	0	100	!= 11111	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit, release	ARMv8.1
10	0	0	1	0	100	11111	STSMAX, STSMAXL — 32-bit, release	ARMv8.1
10	0	0	1	0	101	!= 11111	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit, release	ARMv8.1
10	0	0	1	0	101	11111	STSMIN, STSMINL — 32-bit, release	ARMv8.1
10	0	0	1	0	110	!= 11111	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit, release	ARMv8.1
10	0	0	1	0	110	11111	STUMAX, STUMAXL — 32-bit, release	ARMv8.1
10	0	0	1	0	111	!= 11111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit, release	ARMv8.1
10	0	0	1	0	111	11111	STUMIN, STUMINL — 32-bit, release	ARMv8.1
10	0	0	1	1	000		SWP, SWPA, SWPAL, SWPL — 32-bit, release	ARMv8.1
10	0	1	0	0	000		LDADD, LDADDA, LDADDAL, LDADDL — 32-bit, acquire	ARMv8.1
10	0	1	0	0	001		LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit, acquire	ARMv8.1
10	0	1	0	0	010		LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit, acquire	ARMv8.1
10	0	1	0	0	011		LDSET, LDSETA, LDSETAL, LDSETL — 32-bit, acquire	ARMv8.1
10	0	1	0	0	100		LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit, acquire	ARMv8.1
10	0	1	0	0	101		LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit, acquire	ARMv8.1
10	0	1	0	0	110		LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit, acquire	ARMv8.1
10	0	1	0	0	111		LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit, acquire	ARMv8.1
10	0	1	0	1	000		SWP, SWPA, SWPAL, SWPL — 32-bit, acquire	ARMv8.1
10	0	1	0	1	100		LDAPR — 32-bit	ARMv8.3
10	0	1	1	0	000		LDADD, LDADDA, LDADDAL, LDADDL — 32-bit, acquire and release	ARMv8.1
10	0	1	1	0	001		LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit, acquire and release	ARMv8.1
10	0	1	1	0	010		LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit, acquire and release	ARMv8.1
10	0	1	1	0	011		LDSET, LDSETA, LDSETAL, LDSETL — 32-bit, acquire and release	ARMv8.1
10	0	1	1	0	100		LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit, acquire and release	ARMv8.1
10	0	1	1	0	101		LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit, acquire and release	ARMv8.1
10	0	1	1	0	110		LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit, acquire and release	ARMv8.1
10	0	1	1	0	111		LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit, acquire and release	ARMv8.1
10	0	1	1	1	000		SWP, SWPA, SWPAL, SWPL — 32-bit, acquire and release	ARMv8.1

Decode fields							Instruction Details	Architecture Version
size	V	A	R	o3	opc	Rt		
11	0	0	0	0	000	!= 11111	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	000	11111	STADD, STADDL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	001	!= 11111	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	001	11111	STCLR, STCLRL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	010	!= 11111	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	010	11111	STEOR, STEORL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	011	!= 11111	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	011	11111	STSET, STSETL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	100	!= 11111	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	100	11111	STSMAX, STSMAXL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	101	!= 11111	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	101	11111	STSMIN, STSMINL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	110	!= 11111	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	110	11111	STUMAX, STUMAXL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	111	!= 11111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	0	111	11111	STUMIN, STUMINL — 64-bit, no memory ordering	ARMv8.1
11	0	0	0	1	000		SWP, SWPA, SWPAL, SWPL — 64-bit, no memory ordering	ARMv8.1
11	0	0	1	0	000	!= 11111	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit, release	ARMv8.1
11	0	0	1	0	000	11111	STADD, STADDL — 64-bit, release	ARMv8.1
11	0	0	1	0	001	!= 11111	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit, release	ARMv8.1
11	0	0	1	0	001	11111	STCLR, STCLRL — 64-bit, release	ARMv8.1
11	0	0	1	0	010	!= 11111	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit, release	ARMv8.1
11	0	0	1	0	010	11111	STEOR, STEORL — 64-bit, release	ARMv8.1
11	0	0	1	0	011	!= 11111	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit, release	ARMv8.1
11	0	0	1	0	011	11111	STSET, STSETL — 64-bit, release	ARMv8.1
11	0	0	1	0	100	!= 11111	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit, release	ARMv8.1
11	0	0	1	0	100	11111	STSMAX, STSMAXL — 64-bit, release	ARMv8.1
11	0	0	1	0	101	!= 11111	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit, release	ARMv8.1
11	0	0	1	0	101	11111	STSMIN, STSMINL — 64-bit, release	ARMv8.1
11	0	0	1	0	110	!= 11111	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit, release	ARMv8.1
11	0	0	1	0	110	11111	STUMAX, STUMAXL — 64-bit, release	ARMv8.1
11	0	0	1	0	111	!= 11111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit, release	ARMv8.1

size	V	A	R	o3	opc	Rt	Instruction Details	Architecture Version
11	0	0	1	0	111	11111	STUMIN, STUMINL — 64-bit, release	ARMv8.1
11	0	0	1	1	000		SWP, SWPA, SWPAL, SWPL — 64-bit, release	ARMv8.1
11	0	1	0	0	000		LDADD, LDADDA, LDADDAL, LDADDL — 64-bit, acquire	ARMv8.1
11	0	1	0	0	001		LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit, acquire	ARMv8.1
11	0	1	0	0	010		LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit, acquire	ARMv8.1
11	0	1	0	0	011		LDSET, LDSETA, LDSETAL, LDSETL — 64-bit, acquire	ARMv8.1
11	0	1	0	0	100		LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit, acquire	ARMv8.1
11	0	1	0	0	101		LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit, acquire	ARMv8.1
11	0	1	0	0	110		LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit, acquire	ARMv8.1
11	0	1	0	0	111		LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit, acquire	ARMv8.1
11	0	1	0	1	000		SWP, SWPA, SWPAL, SWPL — 64-bit, acquire	ARMv8.1
11	0	1	0	1	100		LDAPR — 64-bit	ARMv8.3
11	0	1	1	0	000		LDADD, LDADDA, LDADDAL, LDADDL — 64-bit, acquire and release	ARMv8.1
11	0	1	1	0	001		LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit, acquire and release	ARMv8.1
11	0	1	1	0	010		LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit, acquire and release	ARMv8.1
11	0	1	1	0	011		LDSET, LDSETA, LDSETAL, LDSETL — 64-bit, acquire and release	ARMv8.1
11	0	1	1	0	100		LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit, acquire and release	ARMv8.1
11	0	1	1	0	101		LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit, acquire and release	ARMv8.1
11	0	1	1	0	110		LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit, acquire and release	ARMv8.1
11	0	1	1	0	111		LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit, acquire and release	ARMv8.1
11	0	1	1	1	000		SWP, SWPA, SWPAL, SWPL — 64-bit, acquire and release	ARMv8.1

Load/store register (register offset)

These instructions are under Loads and Stores.

31 30	29 28 27	26	25 24	23 22	21	20 19 18 17 16	15 14 13	12	11 10	9 8 7 6 5	4 3 2 1 0
size	1 1 1	V	0 0	opc	1	Rm	option	S	1 0	Rn	Rt

size	V	opc	option	Instruction Details
			x0x	UNALLOCATED
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)

Decode fields size	V	opc	option	Instruction Details
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 size	M	W	Instruction Details	Architecture Version
!= 11			UNALLOCATED	-
11	0	0	LDRAA, LDRAB — key A, offset	ARMv8.3
11	0	1	LDRAA, LDRAB — key A, pre-indexed	ARMv8.3
11	1	0	LDRAA, LDRAB — key B, offset	ARMv8.3
11	1	1	LDRAA, LDRAB — key B, pre-indexed	ARMv8.3

Load/store register (unsigned immediate)

These instructions are under Loads and Stores.

31 30	29 28 27	26	25 24	23 22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
size	1 1 1	V	0 1	opc	imm12	Rn	Rt

Decode fields size	V	opc	Instruction Details
x1	1	1x	UNALLOCATED
00	0	00	STRB (immediate)

Decode fields size	V	opc	Instruction Details
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 op0	op1	op2	op3	Instruction details
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		Add/subtract (with carry)
	1	0010	0	Conditional compare (register)
	1	0010	1	Conditional compare (immediate)
	1	0100		Conditional select
	1	0xx1		UNALLOCATED
	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 sf	S	opcode	Instruction Details	Architecture Version
		00000x	UNALLOCATED	-
		011xxx	UNALLOCATED	-
		1xxxxx	UNALLOCATED	-
	0	0001xx	UNALLOCATED	-
	0	0011xx	UNALLOCATED	-
	1		UNALLOCATED	-
0	0	000010	UDIV — 32-bit	-
0	0	000011	SDIV — 32-bit	-
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	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	000010	UDIV — 64-bit	-
1	0	000011	SDIV — 64-bit	-
1	0	001000	LSLV — 64-bit	-
1	0	001001	LSRV — 64-bit	-
1	0	001010	ASRV — 64-bit	-
1	0	001011	RORV — 64-bit	-
1	0	001100	PACGA	ARMv8.3
1	0	010xx0	UNALLOCATED	-
1	0	010x0x	UNALLOCATED	-
1	0	010011	CRC32B, CRC32H, CRC32W, CRC32X — CRC32X	-
1	0	010111	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX	-

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 sf	S	opcode2	opcode	Rn	Instruction Details	Architecture Version
			xx1xxx		UNALLOCATED	-
			x1xxxx		UNALLOCATED	-
			1xxxxx		UNALLOCATED	-
		xxxx1			UNALLOCATED	-

sf	S	Decode fields opcode2	opcode	Rn	Instruction Details	Architecture Version
		xxx1x			UNALLOCATED	-
		xx1xx			UNALLOCATED	-
		x1xxx			UNALLOCATED	-
		1xxxx			UNALLOCATED	-
	0	00000	00011x		UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00000	000000		RBIT — 32-bit	-
0	0	00000	000001		REV16 — 32-bit	-
0	0	00000	000010		REV — 32-bit	-
0	0	00000	000011		UNALLOCATED	-
0	0	00000	000100		CLZ — 32-bit	-
0	0	00000	000101		CLS — 32-bit	-
1	0	00000	000000		RBIT — 64-bit	-
1	0	00000	000001		REV16 — 64-bit	-
1	0	00000	000010		REV32	-
1	0	00000	000011		REV — 64-bit	-
1	0	00000	000100		CLZ — 64-bit	-
1	0	00000	000101		CLS — 64-bit	-
1	0	00001	000000		PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA	ARMv8.3
1	0	00001	000001		PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB	ARMv8.3
1	0	00001	000010		PACDA, PACDZA — PACDA	ARMv8.3
1	0	00001	000011		PACDB, PACDZB — PACDB	ARMv8.3
1	0	00001	000100		AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA	ARMv8.3
1	0	00001	000101		AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB	ARMv8.3
1	0	00001	000110		AUTDA, AUTDZA — AUTDA	ARMv8.3
1	0	00001	000111		AUTDB, AUTDZB — AUTDB	ARMv8.3
1	0	00001	001000	11111	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIZA	ARMv8.3
1	0	00001	001001	11111	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIZB	ARMv8.3
1	0	00001	001010	11111	PACDA, PACDZA — PACDZA	ARMv8.3
1	0	00001	001011	11111	PACDB, PACDZB — PACDZB	ARMv8.3
1	0	00001	001100	11111	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIZA	ARMv8.3
1	0	00001	001101	11111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIZB	ARMv8.3
1	0	00001	001110	11111	AUTDA, AUTDZA — AUTDZA	ARMv8.3
1	0	00001	001111	11111	AUTDB, AUTDZB — AUTDZB	ARMv8.3
1	0	00001	010000	11111	XPACD, XPACI, XPACLRI — XPACI	ARMv8.3
1	0	00001	010001	11111	XPACD, XPACI, XPACLRI — XPACD	ARMv8.3

Logical (shifted register)

These instructions are under Data Processing -- Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc		0	1	0	1	0	shift		N	Rm					imm6						Rn					Rd				

Decode fields				Instruction Details
sf	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

Decode fields					Instruction Details
sf	op	S	opt	imm3	
			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	opcode2	Rn	Rd

Decode fields				Instruction Details
sf	op	S	opcode2	
			xxxxx1	UNALLOCATED
			xxxx1x	UNALLOCATED
			xxx1xx	UNALLOCATED
			xx1xxx	UNALLOCATED
			x1xxxx	UNALLOCATED
			1xxxxx	UNALLOCATED
0	0	0	000000	ADC — 32-bit
0	0	1	000000	ADCS — 32-bit
0	1	0	000000	SBC — 32-bit
0	1	1	000000	SBCS — 32-bit
1	0	0	000000	ADC — 64-bit
1	0	1	000000	ADCS — 64-bit
1	1	0	000000	SBC — 64-bit
1	1	1	000000	SBCS — 64-bit

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
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	nzcv

Decode fields sf	op	S	o2	o3	Instruction Details
				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 sf	op	S	op2	Instruction Details
			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 sf	op54	op31	o0	Instruction Details
	00	010	1	UNALLOCATED
	00	011		UNALLOCATED
	00	100		UNALLOCATED
	00	110	1	UNALLOCATED
	00	111		UNALLOCATED
	01			UNALLOCATED
	1x			UNALLOCATED
0	00	000	0	MADD — 32-bit
0	00	000	1	MSUB — 32-bit

Decode fields				Instruction Details
sf	op54	op31	o0	
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	xxx0xxxx0	UNALLOCATED	-
0111	0x	x101	00xxxxx10	UNALLOCATED	-
01x1	00	00xx	xxx0xxxx1	Advanced SIMD scalar copy	-
01x1	01	00xx	xxx0xxxx1	UNALLOCATED	-
01x1	0x	0111	00xxxxx10	UNALLOCATED	-
01x1	0x	10xx	xxx00xxx1	Advanced SIMD scalar three same FP16	ARMv8.2
01x1	0x	10xx	xxx01xxx1	UNALLOCATED	-
01x1	0x	1111	00xxxxx10	Advanced SIMD scalar two-register miscellaneous FP16	ARMv8.2
01x1	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
01x1	0x	x0xx	xxx1xxxx1	Advanced SIMD scalar three same extra	ARMv8.1
01x1	0x	x100	00xxxxx10	Advanced SIMD scalar two-register miscellaneous	-
01x1	0x	x110	00xxxxx10	Advanced SIMD scalar pairwise	ARMv8.2
01x1	0x	x1xx	1xxxxxx10	UNALLOCATED	-
01x1	0x	x1xx	x1xxxxx10	UNALLOCATED	-
01x1	0x	x1xx	xxxxxxx00	Advanced SIMD scalar three different	-
01x1	0x	x1xx	xxxxxxxx1	Advanced SIMD scalar three same	-
01x1	10		xxxxxxxx1	Advanced SIMD scalar shift by immediate	-

01x1	11		xxxxxxxx1	UNALLOCATED	-
01x1	1x		xxxxxxxx0	Advanced SIMD scalar x indexed element	ARMv8.2
0x00	0x	x0xx	xxx0xxx00	Advanced SIMD table lookup	-
0x00	0x	x0xx	xxx0xxx10	Advanced SIMD permute	-
0x10	0x	x0xx	xxx0xxxx0	Advanced SIMD extract	-
0xx0	00	00xx	xxx0xxxx1	Advanced SIMD copy	-
0xx0	01	00xx	xxx0xxxx1	UNALLOCATED	-
0xx0	0x	0111	00xxxxx10	UNALLOCATED	-
0xx0	0x	10xx	xxx00xxx1	Advanced SIMD three same (FP16)	ARMv8.2
0xx0	0x	10xx	xxx01xxx1	UNALLOCATED	-
0xx0	0x	1111	00xxxxx10	Advanced SIMD two-register miscellaneous (FP16)	ARMv8.2
0xx0	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0xx0	0x	x0xx	xxx1xxxx1	Advanced SIMD three same extra	ARMv8.2
0xx0	0x	x100	00xxxxx10	Advanced SIMD two-register miscellaneous	-
0xx0	0x	x110	00xxxxx10	Advanced SIMD across lanes	ARMv8.2
0xx0	0x	x1xx	1xxxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	x1xxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	xxxxxxx00	Advanced SIMD three different	-
0xx0	0x	x1xx	xxxxxxxx1	Advanced SIMD three same	-
0xx0	10	0000	xxxxxxxx1	Advanced SIMD modified immediate	ARMv8.2
0xx0	10	!= 0000	xxxxxxxx1	Advanced SIMD shift by immediate	-
0xx0	11		xxxxxxxx1	UNALLOCATED	-
0xx0	1x		xxxxxxxx0	Advanced SIMD vector x indexed element	ARMv8.2
1100	00	10xx	xxx10xxxx	Cryptographic three-register, imm2	ARMv8.2
1100	00	11xx	xxx1x00xx	Cryptographic three-register SHA 512	ARMv8.2
1100	00		xxx0xxxxx	Cryptographic four-register	ARMv8.2
1100	01	00xx		XAR	ARMv8.2
1100	01	1000	0001000xx	Cryptographic two-register SHA 512	ARMv8.2
11x1				UNALLOCATED	-
1xx0	1x			UNALLOCATED	-
x0x1	0x	x0xx		Conversion between floating-point and fixed-point	ARMv8.2
x0x1	0x	x1xx	xxx000000	Conversion between floating-point and integer	ARMv8.3
x0x1	0x	x1xx	xxx100000	UNALLOCATED	-
x0x1	0x	x1xx	xxxx10000	Floating-point data-processing (1 source)	ARMv8.2
x0x1	0x	x1xx	xxxxx1000	Floating-point compare	ARMv8.2
x0x1	0x	x1xx	xxxxxx100	Floating-point immediate	ARMv8.2
x0x1	0x	x1xx	xxxxxxx01	Floating-point conditional compare	ARMv8.2
x0x1	0x	x1xx	xxxxxxx10	Floating-point data-processing (2 source)	ARMv8.2
x0x1	0x	x1xx	xxxxxxx11	Floating-point conditional select	ARMv8.2
x0x1	1x			Floating-point data-processing (3 source)	ARMv8.2

Cryptographic AES

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23 22	21	20	19	18	17	16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	0	0	1	1	1	0	size	1	0	1	0	0	opcode	1	0	Rn	Rd

Decode fields size	opcode	Instruction Details
	x1xxx	UNALLOCATED
	000xx	UNALLOCATED
	1xxxx	UNALLOCATED
x1		UNALLOCATED
00	00100	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
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				

op	Decode fields imm5	imm4	Instruction Details
0		xxx1	UNALLOCATED
0		xx1x	UNALLOCATED
0		x1xx	UNALLOCATED
0		0000	DUP (element)
0		1xxx	UNALLOCATED
0	x0000	0000	UNALLOCATED
1			UNALLOCATED

Advanced SIMD scalar three same FP16

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	a	1	0	Rm					0	0	opcode			1	Rn					Rd				

U	Decode fields a	opcode	Instruction Details	Architecture Version
		110	UNALLOCATED	-
	1	011	UNALLOCATED	-
0	0	011	FMULX	ARMv8.2
0	0	100	FCMEQ (register)	ARMv8.2
0	0	101	UNALLOCATED	-
0	0	111	FRECPS	ARMv8.2
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	111	FRSQRTS	ARMv8.2
1	0	011	UNALLOCATED	-
1	0	100	FCMGE (register)	ARMv8.2
1	0	101	FACGE	ARMv8.2
1	0	111	UNALLOCATED	-
1	1	010	FABD	ARMv8.2
1	1	100	FCMGT (register)	ARMv8.2
1	1	101	FACGT	ARMv8.2
1	1	111	UNALLOCATED	-

Advanced SIMD scalar two-register miscellaneous FP16

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	a	1	1	1	1	0	0	opcode					1	0	Rn					Rd				

U	Decode fields a	opcode	Instruction Details	Architecture Version
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-

Decode fields U	a	opcode	Instruction Details	Architecture Version
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	01111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11010	FCVTNS (vector)	ARMv8.2
0	0	11011	FCVTMS (vector)	ARMv8.2
0	0	11100	FCVTAS (vector)	ARMv8.2
0	0	11101	SCVTF (vector, integer)	ARMv8.2
0	1	01100	FCMGT (zero)	ARMv8.2
0	1	01101	FCMEQ (zero)	ARMv8.2
0	1	01110	FCMLT (zero)	ARMv8.2
0	1	11010	FCVTPS (vector)	ARMv8.2
0	1	11011	FCVTZS (vector, integer)	ARMv8.2
0	1	11101	FRECPE	ARMv8.2
0	1	11111	FRECPX	ARMv8.2
1	0	11010	FCVTNU (vector)	ARMv8.2
1	0	11011	FCVTMU (vector)	ARMv8.2
1	0	11100	FCVTAU (vector)	ARMv8.2
1	0	11101	UCVTF (vector, integer)	ARMv8.2
1	1	01100	FCMGE (zero)	ARMv8.2
1	1	01101	FCMLE (zero)	ARMv8.2
1	1	01110	UNALLOCATED	-
1	1	11010	FCVTPU (vector)	ARMv8.2
1	1	11011	FCVTZU (vector, integer)	ARMv8.2
1	1	11101	FRSQRTE	ARMv8.2
1	1	11111	UNALLOCATED	-

Advanced SIMD scalar three same extra

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size		0	Rm					1	opcode				1	Rn						Rd			

Decode fields U	opcode	Instruction Details	Architecture Version
	001x	UNALLOCATED	-
	01xx	UNALLOCATED	-
	1xxx	UNALLOCATED	-
0	0000	UNALLOCATED	-
0	0001	UNALLOCATED	-
1	0000	SQRDMLAH (vector)	ARMv8.1
1	0001	SQRDMLSH (vector)	ARMv8.1

Advanced SIMD scalar two-register miscellaneous

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size		1	0	0	0	0	opcode					1	0	Rn					Rd				

Decode fields U	size	opcode	Instruction Details
		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 U	size	opcode	Instruction Details
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	FRSQRTE
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 U	size	opcode	Instruction Details	Architecture Version
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11010	UNALLOCATED	-
		111xx	UNALLOCATED	-
	1x	01101	UNALLOCATED	-
0		11011	ADDP (scalar)	-
0	00	01100	FMAXNMP (scalar) — half-precision	ARMv8.2
0	00	01101	FADDP (scalar) — half-precision	ARMv8.2
0	00	01111	FMAXP (scalar) — half-precision	ARMv8.2
0	01	01100	UNALLOCATED	-
0	01	01101	UNALLOCATED	-
0	01	01111	UNALLOCATED	-
0	10	01100	FMINNMP (scalar) — half-precision	ARMv8.2
0	10	01111	FMINP (scalar) — half-precision	ARMv8.2
0	11	01100	UNALLOCATED	-
0	11	01111	UNALLOCATED	-
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 U	opcode	Instruction Details
	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	opcode	1	Rn	Rd

Decode fields U	size	opcode	Instruction Details
		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

Decode fields			Instruction Details
U	size	opcode	
0	0x	11010	UNALLOCATED
0	0x	11011	FMULX
0	0x	11100	FCMEQ (register)
0	0x	11101	UNALLOCATED
0	0x	11110	UNALLOCATED
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 immh	opcode	Instruction Details
	!= 0000	00001	UNALLOCATED
	!= 0000	00011	UNALLOCATED
	!= 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	SRSHR
0	!= 0000	00110	SRSRA
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 U	size	opcode	Instruction Details	Architecture Version
		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	ARMv8.2
0	00	0101	FMLS (by element) — half-precision	ARMv8.2
0	00	1001	FMUL (by element) — half-precision	ARMv8.2
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)	ARMv8.1
1		1111	SQRDMLSH (by element)	ARMv8.1
1	00	0001	UNALLOCATED	-
1	00	0101	UNALLOCATED	-
1	00	1001	FMULX (by element) — half-precision	ARMv8.2
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 op2	len	op	Instruction Details
x1			UNALLOCATED
00	00	0	TBL — single register table
00	00	1	TBX — single register table

Decode fields op2	len	op	Instruction Details
00	01	0	TBL — two register table
00	01	1	TBX — two register table
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					Rd				

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					Rd				

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					Rd				

Decode fields Q	op	imm5	imm4	Instruction Details
		x0000		UNALLOCATED
	0		0000	DUP (element)
	0		0001	DUP (general)
	0		0010	UNALLOCATED

Decode fields				Instruction Details
Q	op	imm5	imm4	
	0		0100	UNALLOCATED
	0		0110	UNALLOCATED
	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	Architecture Version
U	a	opcode		
0	0	000	FMAXNM (vector)	ARMv8.2
0	0	001	FMLA (vector)	ARMv8.2
0	0	010	FADD (vector)	ARMv8.2
0	0	011	FMULX	ARMv8.2
0	0	100	FCMEQ (register)	ARMv8.2
0	0	101	UNALLOCATED	-
0	0	110	FMAX (vector)	ARMv8.2
0	0	111	FRECPS	ARMv8.2
0	1	000	FMINNM (vector)	ARMv8.2
0	1	001	FMLS (vector)	ARMv8.2
0	1	010	FSUB (vector)	ARMv8.2
0	1	011	UNALLOCATED	-
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	110	FMIN (vector)	ARMv8.2
0	1	111	FRSQRTS	ARMv8.2
1	0	000	FMAXNMP (vector)	ARMv8.2
1	0	001	UNALLOCATED	-
1	0	010	FADDP (vector)	ARMv8.2
1	0	011	FMUL (vector)	ARMv8.2
1	0	100	FCMGE (register)	ARMv8.2
1	0	101	FACGE	ARMv8.2
1	0	110	FMAXP (vector)	ARMv8.2
1	0	111	FDIV (vector)	ARMv8.2
1	1	000	FMINNMP (vector)	ARMv8.2
1	1	001	UNALLOCATED	-
1	1	010	FABD	ARMv8.2
1	1	011	UNALLOCATED	-

Decode fields U	a	opcode	Instruction Details	Architecture Version
1	1	100	FCMGT (register)	ARMv8.2
1	1	101	FACGT	ARMv8.2
1	1	110	FMINP (vector)	ARMv8.2
1	1	111	UNALLOCATED	-

Advanced SIMD two-register miscellaneous (FP16)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16 15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
0	Q	U	0	1	1	1	0	a	1	1	1	1	0	0	opcode	1	0	Rn	Rd

Decode fields U	a	opcode	Instruction Details	Architecture Version
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11000	FRINTN (vector)	ARMv8.2
0	0	11001	FRINTM (vector)	ARMv8.2
0	0	11010	FCVTNS (vector)	ARMv8.2
0	0	11011	FCVTMS (vector)	ARMv8.2
0	0	11100	FCVTAS (vector)	ARMv8.2
0	0	11101	SCVTF (vector, integer)	ARMv8.2
0	1	01100	FCMGT (zero)	ARMv8.2
0	1	01101	FCMEQ (zero)	ARMv8.2
0	1	01110	FCMLT (zero)	ARMv8.2
0	1	01111	FABS (vector)	ARMv8.2
0	1	11000	FRINTP (vector)	ARMv8.2
0	1	11001	FRINTZ (vector)	ARMv8.2
0	1	11010	FCVTPS (vector)	ARMv8.2
0	1	11011	FCVTZS (vector, integer)	ARMv8.2
0	1	11101	FRECPE	ARMv8.2
0	1	11111	UNALLOCATED	-
1	0	11000	FRINTA (vector)	ARMv8.2
1	0	11001	FRINTX (vector)	ARMv8.2
1	0	11010	FCVTNU (vector)	ARMv8.2
1	0	11011	FCVTMU (vector)	ARMv8.2
1	0	11100	FCVTAU (vector)	ARMv8.2
1	0	11101	UCVTF (vector, integer)	ARMv8.2
1	1	01100	FCMGE (zero)	ARMv8.2
1	1	01101	FCMLE (zero)	ARMv8.2
1	1	01110	UNALLOCATED	-
1	1	01111	FNEG (vector)	ARMv8.2
1	1	11000	UNALLOCATED	-
1	1	11001	FRINTI (vector)	ARMv8.2

Decode fields U	a	opcode	Instruction Details	Architecture Version
1	1	11010	FCVTPU (vector)	ARMv8.2
1	1	11011	FCVTZU (vector, integer)	ARMv8.2
1	1	11101	FRSQRTE	ARMv8.2
1	1	11111	FSQRT (vector)	ARMv8.2

Advanced SIMD three same extra

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size		0	Rm					1	opcode				1	Rn					Rd				

Decode fields U	opcode	Instruction Details	Architecture Version
	0011	UNALLOCATED	-
	01xx	UNALLOCATED	-
0	0000	UNALLOCATED	-
0	0001	UNALLOCATED	-
0	0010	SDOT (vector)	ARMv8.2
0	1xxx	UNALLOCATED	-
1	0000	SQRDMLAH (vector)	ARMv8.1
1	0001	SQRDMLSH (vector)	ARMv8.1
1	0010	UDOT (vector)	ARMv8.2
1	10xx	FCMLA	ARMv8.3
1	11x0	FCADD	ARMv8.3
1	11x1	UNALLOCATED	-

Advanced SIMD two-register miscellaneous

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size		1	0	0	0	0	opcode					1	0	Rn					Rd				

Decode fields U	size	opcode	Instruction Details
		1000x	UNALLOCATED
		10101	UNALLOCATED
		11110	UNALLOCATED
	0x	011xx	UNALLOCATED
	0x	11111	UNALLOCATED
	1x	10110	UNALLOCATED
	1x	10111	UNALLOCATED
0		00000	REV64
0		00001	REV16 (vector)
0		00010	SADDLP
0		00011	SUQADD
0		00100	CLS (vector)
0		00101	CNT
0		00110	SADALP

Decode fields			Instruction Details
U	size	opcode	
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	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
1		00000	REV32 (vector)
1		00001	UNALLOCATED
1		00010	UADDLP
1		00011	USQADD
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)

Decode fields U	size	opcode	Instruction Details
1	0x	11100	FCVTAU (vector)
1	0x	11101	UCVTF (vector, integer)
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)

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 U	size	opcode	Instruction Details	Architecture Version
		0000x	UNALLOCATED	-
		00010	UNALLOCATED	-
		001xx	UNALLOCATED	-
		0100x	UNALLOCATED	-
		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	ARMv8.2
0	00	01111	FMAXV — half-precision	ARMv8.2
0	01	01100	UNALLOCATED	-
0	01	01111	UNALLOCATED	-
0	10	01100	FMINNMV — half-precision	ARMv8.2
0	10	01111	FMINV — half-precision	ARMv8.2
0	11	01100	UNALLOCATED	-
0	11	01111	UNALLOCATED	-
1		00011	UADDLV	-

Decode fields U	size	opcode	Instruction Details	Architecture Version
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 U	opcode	Instruction Details
	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)
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 U	size	opcode	Instruction Details
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
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	11101	UNALLOCATED
0	0x	11110	FMAX (vector)
0	0x	11111	FRECPS
0	00	00011	AND (vector)
0	01	00011	BIC (vector, register)
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	11101	UNALLOCATED
0	1x	11110	FMIN (vector)
0	1x	11111	FRSQRTS
0	10	00011	ORR (vector, register)

Decode fields			Instruction Details
U	size	opcode	
0	11	00011	ORN (vector)
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
1	0x	11000	FMAXNMP (vector)
1	0x	11001	UNALLOCATED
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	01	00011	BSL
1	1x	11000	FMINNMP (vector)
1	1x	11001	UNALLOCATED
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	11	00011	BIF

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	op	cmode	o2	Instruction Details	Architecture Version
	0	0xxx	1	UNALLOCATED	-
	0	0xx0	0	MOVI — 32-bit shifted immediate	-
	0	0xx1	0	ORR (vector, immediate) — 32-bit	-
	0	10xx	1	UNALLOCATED	-
	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	ARMv8.2
	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	-
	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 immh	immb	opcode		1	Rn	Rd

The following constraints also apply to this encoding: immh != 0000 && immh != 0000

U	opcode	Instruction Details
	00001	UNALLOCATED
	00011	UNALLOCATED
	00101	UNALLOCATED
	00111	UNALLOCATED
	01001	UNALLOCATED
	01011	UNALLOCATED
	01101	UNALLOCATED
	01111	UNALLOCATED
	10101	UNALLOCATED
	1011x	UNALLOCATED
	110xx	UNALLOCATED
	11101	UNALLOCATED

Decode fields U	opcode	Instruction Details
	11110	UNALLOCATED
0	00000	SSHR
0	00010	SSRA
0	00100	SRSHR
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
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 U	size	opcode	Instruction Details	Architecture Version
	01	1001	UNALLOCATED	-
0		0000	UNALLOCATED	-
0		0010	SMLAL, SMLAL2 (by element)	-
0		0011	SQDMLAL, SQDMLAL2 (by element)	-
0		0100	UNALLOCATED	-
0		0110	SMLSL, SMLSL2 (by element)	-
0		0111	SQDMLSL, SQDMLSL2 (by element)	-
0		1000	MUL (by element)	-

Decode fields U	size	opcode	Instruction Details	Architecture Version
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)	ARMv8.2
0		1111	UNALLOCATED	-
0	00	0001	FMLA (by element) — half-precision	ARMv8.2
0	00	0101	FMLS (by element) — half-precision	ARMv8.2
0	00	1001	FMUL (by element) — half-precision	ARMv8.2
0	01	0001	UNALLOCATED	-
0	01	0101	UNALLOCATED	-
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		0000	MLA (by element)	-
1		0010	UMLAL, UMLAL2 (by element)	-
1		0100	MLS (by element)	-
1		0110	UMLSL, UMLSL2 (by element)	-
1		1000	UNALLOCATED	-
1		1010	UMULL, UMULL2 (by element)	-
1		1011	UNALLOCATED	-
1		1100	UNALLOCATED	-
1		1101	SQRDMLAH (by element)	ARMv8.1
1		1110	UDOT (by element)	ARMv8.2
1		1111	SQRDMLSH (by element)	ARMv8.1
1	00	0001	UNALLOCATED	-
1	00	0011	UNALLOCATED	-
1	00	0101	UNALLOCATED	-
1	00	0111	UNALLOCATED	-
1	00	1001	FMULX (by element) — half-precision	ARMv8.2
1	01	0xx1	FCMLA (by element)	ARMv8.3
1	1x	0101	UNALLOCATED	-
1	1x	1001	FMULX (by element) — single-precision and double-precision	-
1	10	0xx1	FCMLA (by element)	ARMv8.3
1	11	0001	UNALLOCATED	-
1	11	0011	UNALLOCATED	-
1	11	0111	UNALLOCATED	-

Cryptographic three-register, imm2

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15	14	13 12	11 10	9 8 7 6 5	4 3 2 1 0
1	1	0	0	1	1	1	0	0	1	0	Rm	1	0	imm2	opcode	Rn	Rd

Decode fields opcode	Instruction Details	Architecture Version
00	SM3TT1A	ARMv8.2
01	SM3TT1B	ARMv8.2

Decode fields opcode	Instruction Details	Architecture Version
10	SM3TT2A	ARMv8.2
11	SM3TT2B	ARMv8.2

Cryptographic three-register SHA 512

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm					1	O	0	0	opcode		Rn					Rd				

Decode fields O	opcode	Instruction Details	Architecture Version
0	00	SHA512H	ARMv8.2
0	01	SHA512H2	ARMv8.2
0	10	SHA512SU1	ARMv8.2
0	11	RAX1	ARMv8.2
1	00	SM3PARTW1	ARMv8.2
1	01	SM3PARTW2	ARMv8.2
1	10	SM4EKEY	ARMv8.2
1	11	UNALLOCATED	-

Cryptographic four-register

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	Op0		Rm					0	Ra					Rn					Rd				

Decode fields Op0	Instruction Details	Architecture Version
00	EOR3	ARMv8.2
01	BCAX	ARMv8.2
10	SM3SS1	ARMv8.2
11	UNALLOCATED	-

Cryptographic two-register SHA 512

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	opcode		Rn					Rd				

Decode fields opcode	Instruction Details	Architecture Version
00	SHA512SU0	ARMv8.2
01	SM4E	ARMv8.2
1x	UNALLOCATED	-

Conversion between floating-point and fixed-point

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	S	1	1	1	1	0	type		0	rmode		opcode			scale						Rn					Rd				

sf	S	type	rmode	opcode	scale	Instruction Details	Architecture Version
				1xx		UNALLOCATED	-
			x0	00x		UNALLOCATED	-
			x1	01x		UNALLOCATED	-
			0x	00x		UNALLOCATED	-
			1x	01x		UNALLOCATED	-
		10				UNALLOCATED	-
	1					UNALLOCATED	-
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	ARMv8.2
0	0	11	00	011		UCVTF (scalar, fixed-point) — 32-bit to half-precision	ARMv8.2
0	0	11	11	000		FCVTZS (scalar, fixed-point) — half-precision to 32-bit	ARMv8.2
0	0	11	11	001		FCVTZU (scalar, fixed-point) — half-precision to 32-bit	ARMv8.2
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	ARMv8.2
1	0	11	00	011		UCVTF (scalar, fixed-point) — 64-bit to half-precision	ARMv8.2
1	0	11	11	000		FCVTZS (scalar, fixed-point) — half-precision to 64-bit	ARMv8.2
1	0	11	11	001		FCVTZU (scalar, fixed-point) — half-precision to 64-bit	ARMv8.2

Conversion between floating-point and integer

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	S	1	1	1	1	0	type		1	rmode		opcode			0	0	0	0	0	0	Rn					Rd				

sf	S	Decode fields type	rmode	opcode	Instruction Details	Architecture Version
			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	FJCVTZS	ARMv8.3
0	0	01	11	111	UNALLOCATED	-
0	0	10		11x	UNALLOCATED	-
0	0	11	00	000	FCVTNS (scalar) — half-precision to 32-bit	ARMv8.2
0	0	11	00	001	FCVTNU (scalar) — half-precision to 32-bit	ARMv8.2
0	0	11	00	010	SCVTF (scalar, integer) — 32-bit to half-precision	ARMv8.2
0	0	11	00	011	UCVTF (scalar, integer) — 32-bit to half-precision	ARMv8.2
0	0	11	00	100	FCVTAS (scalar) — half-precision to 32-bit	ARMv8.2
0	0	11	00	101	FCVTAU (scalar) — half-precision to 32-bit	ARMv8.2
0	0	11	00	110	FMOV (general) — half-precision to 32-bit	ARMv8.2

sf	S	Decode fields type	rmode	opcode	Instruction Details	Architecture Version
0	0	11	00	111	FMOV (general) — 32-bit to half-precision	ARMv8.2
0	0	11	01	000	FCVTPS (scalar) — half-precision to 32-bit	ARMv8.2
0	0	11	01	001	FCVTPU (scalar) — half-precision to 32-bit	ARMv8.2
0	0	11	10	000	FCVTMS (scalar) — half-precision to 32-bit	ARMv8.2
0	0	11	10	001	FCVTMU (scalar) — half-precision to 32-bit	ARMv8.2
0	0	11	11	000	FCVTZS (scalar, integer) — half-precision to 32-bit	ARMv8.2
0	0	11	11	001	FCVTZU (scalar, integer) — half-precision to 32-bit	ARMv8.2
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	ARMv8.2
1	0	11	00	001	FCVTNU (scalar) — half-precision to 64-bit	ARMv8.2
1	0	11	00	010	SCVTF (scalar, integer) — 64-bit to half-precision	ARMv8.2
1	0	11	00	011	UCVTF (scalar, integer) — 64-bit to half-precision	ARMv8.2
1	0	11	00	100	FCVTAS (scalar) — half-precision to 64-bit	ARMv8.2
1	0	11	00	101	FCVTAU (scalar) — half-precision to 64-bit	ARMv8.2
1	0	11	00	110	FMOV (general) — half-precision to 64-bit	ARMv8.2

sf	S	Decode fields type	rmode	opcode	Instruction Details	Architecture Version
1	0	11	00	111	FMOV (general) — 64-bit to half-precision	ARMv8.2
1	0	11	01	000	FCVTPS (scalar) — half-precision to 64-bit	ARMv8.2
1	0	11	01	001	FCVTPU (scalar) — half-precision to 64-bit	ARMv8.2
1	0	11	10	000	FCVTMS (scalar) — half-precision to 64-bit	ARMv8.2
1	0	11	10	001	FCVTMU (scalar) — half-precision to 64-bit	ARMv8.2
1	0	11	11	000	FCVTZS (scalar, integer) — half-precision to 64-bit	ARMv8.2
1	0	11	11	001	FCVTZU (scalar, integer) — half-precision to 64-bit	ARMv8.2

Floating-point data-processing (1 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	type		1	opcode						1	0	0	0	0	Rn					Rd				

M	S	Decode fields type	opcode	Instruction Details	Architecture Version
			x1xxxx	UNALLOCATED	-
			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	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	UNALLOCATED	-
0	0	01	000111	FCVT — double-precision to half-precision	-
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	-

Decode fields				Instruction Details	Architecture Version
M	S	type	opcode		
0	0	01	001101	UNALLOCATED	-
0	0	01	001110	FRINTX (scalar) — double-precision	-
0	0	01	001111	FRINTI (scalar) — double-precision	-
0	0	10	00xxxx	UNALLOCATED	-
0	0	11	000000	FMOV (register) — half-precision	ARMv8.2
0	0	11	000001	FABS (scalar) — half-precision	ARMv8.2
0	0	11	000010	FNEG (scalar) — half-precision	ARMv8.2
0	0	11	000011	FSQRT (scalar) — half-precision	ARMv8.2
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	ARMv8.2
0	0	11	001001	FRINTP (scalar) — half-precision	ARMv8.2
0	0	11	001010	FRINTM (scalar) — half-precision	ARMv8.2
0	0	11	001011	FRINTZ (scalar) — half-precision	ARMv8.2
0	0	11	001100	FRINTA (scalar) — half-precision	ARMv8.2
0	0	11	001101	UNALLOCATED	-
0	0	11	001110	FRINTX (scalar) — half-precision	ARMv8.2
0	0	11	001111	FRINTI (scalar) — half-precision	ARMv8.2
1				UNALLOCATED	-

Floating-point compare

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	type		1	Rm					op		1	0	0	0	Rn					opcode2				

Decode fields					Instruction Details	Architecture Version
M	S	type	op	opcode2		
				xxxx1	UNALLOCATED	-
				xxx1x	UNALLOCATED	-
				xx1xx	UNALLOCATED	-
			x1		UNALLOCATED	-
			1x		UNALLOCATED	-
		10			UNALLOCATED	-
	1				UNALLOCATED	-
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	ARMv8.2
0	0	11	00	01000	FCMP	ARMv8.2
0	0	11	00	10000	FCMPE	ARMv8.2
0	0	11	00	11000	FCMPE	ARMv8.2

M	S	type	op	opcode2	Instruction Details	Architecture Version
1					UNALLOCATED	-

Floating-point immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13	12	11	10	9 8 7 6 5	4 3 2 1 0
M	0	S	1	1	1	1	0	type		1	imm8		1	0	0	imm5	Rd

M	S	type	imm5	Instruction Details	Architecture Version
			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	ARMv8.2
1				UNALLOCATED	-

Floating-point conditional compare

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12	11	10	9 8 7 6 5	4	3 2 1 0
M	0	S	1	1	1	1	0	type		1	Rm	cond	0	1	Rn	op	nzcv

M	S	type	op	Instruction Details	Architecture Version
		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	-
0	0	11	0	FCCMP — half-precision	ARMv8.2
0	0	11	1	FCCMPE — half-precision	ARMv8.2
1				UNALLOCATED	-

Floating-point data-processing (2 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20 19 18 17 16	15 14 13 12	11	10	9 8 7 6 5	4 3 2 1 0
M	0	S	1	1	1	1	0	type		1	Rm	opcode	1	0	Rn	Rd

M	S	type	opcode	Instruction Details	Architecture Version
			1xx1	UNALLOCATED	-

Decode fields M	S	type	opcode	Instruction Details	Architecture Version
			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	ARMv8.2
0	0	11	0001	FDIV (scalar) — half-precision	ARMv8.2
0	0	11	0010	FADD (scalar) — half-precision	ARMv8.2
0	0	11	0011	FSUB (scalar) — half-precision	ARMv8.2
0	0	11	0100	FMAX (scalar) — half-precision	ARMv8.2
0	0	11	0101	FMIN (scalar) — half-precision	ARMv8.2
0	0	11	0110	FMAXNM (scalar) — half-precision	ARMv8.2
0	0	11	0111	FMINNM (scalar) — half-precision	ARMv8.2
0	0	11	1000	FNMUL (scalar) — half-precision	ARMv8.2
1				UNALLOCATED	-

Floating-point conditional select

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	type		1	Rm					cond				1	1	Rn					Rd				

Decode fields M	S	type	Instruction Details	Architecture Version
		10	UNALLOCATED	-
	1		UNALLOCATED	-
0	0	00	FCSEL — single-precision	-
0	0	01	FCSEL — double-precision	-
0	0	11	FCSEL — half-precision	ARMv8.2
1			UNALLOCATED	-

Floating-point data-processing (3 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	1	type		o1	Rm					o0	Ra					Rn					Rd				

Decode fields					Instruction Details	Architecture Version
M	S	type	o1	o0		
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	0	0	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	ARMv8.2
0	0	11	0	1	FMSUB — half-precision	ARMv8.2
0	0	11	1	0	FNMADD — half-precision	ARMv8.2
0	0	11	1	1	FNMSUB — half-precision	ARMv8.2
1					UNALLOCATED	-

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages.

Pseudocodes

Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

```
// AArch32.VCRMatch()
// ==================

boolean AArch32.VCRMatch(bits(32) vaddress)

    if UsingAArch32() && ELUsingAArch32(EL1) && IsZero(vaddress<1:0>) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && !IsSecure() then
                match_word<UInt(vaddress<4:2>) + 24> = '1';     // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1';      // Secure vectors (or no EL3)

        if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() && vaddress<31:5> == MVBAR<31:5> then
            match_word<UInt(vaddress<4:2>) + 8> = '1';          // Monitor vectors

        // Mask out bits not corresponding to vectors.
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
        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);
    else
        match = FALSE;

    return match;
```

Library pseudocode for aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// ========================================================

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;
```

Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIDR.BRPs);

    enabled = DBGBCR[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then                  // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        if !value_mismatch then value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHAL

    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
```

Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointValueMatch

```
// 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-existant breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(DBGDIDR.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(DBGDIDR.BRPs), Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking.)
    if DBGBCR[n].E == '0' then return (FALSE,FALSE);

    context_aware = (n >= UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    type = DBGBCR[n].BT;

    if ((type IN {'011x','11xx'} && !HaveVirtHostExt()) ||              // Context matching
          (type == '010x' && HaltOnBreakpointOrWatchpoint()) ||         // Address mismatch
          (type != '0x0x' && !context_aware) ||                         // Context matching
          (type == '1xxx' && !HaveEL(EL2))) then                        // EL2 extension
        (c, type) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = (type == '0x0x');
    mismatch   = (type == '010x');
    match_vmid = (type == '10xx');
    match_cid1 = (type == 'xx1x');
    match_cid2 = (type == '11xx');
    linked     = (type == 'xxx1');

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return (FALSE,FALSE);

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return (FALSE,FALSE);

    // Do the comparison.
    if match_addr then
        byte = UInt(vaddress<1:0>);
        assert byte IN {0,2};                      // "vaddress" is halfword aligned.
        byte_select_match = (DBGBCR[n].BAS<byte> == '1');
        BVR_match = vaddress<31:2> == DBGBVR[n]<31:2> && byte_select_match;
    elsif match_cid1 then
        BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
    if match_vmid then
        if ELUsingAArch32(EL2) then
            vmid = ZeroExtend(VTTBR.VMID, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        else
```

```
            vmid = VTTBR_EL2.VMID;
            bvr_vmid = DBGBXVR[n]<15:0>;
        BXVR_match = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                      vmid == bvr_vmid);
    elsif match_cid2 then
        BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
                      !ELUsingAArch32(EL2) &&
                      DBGBXVR[n]<31:0> == CONTEXTIDR_EL2);

    bvr_match_valid = (match_addr || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

    return (match && !mismatch, !match && mismatch);
```

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
    if ((HMC:SSC:PxC) IN {'011xx','100x0','101x0','11010','11101','1111x'} ||       // Reserved
          (HMC == '0' && PxC == '00' && !isbreakpnt) ||                             // Usr/Svc/Sys
          (SSC IN {'01','10'} && !HaveEL(EL3)) ||                                   // No EL3
          (HMC:SSC:PxC == '11000' && ELUsingAArch32(EL3)) ||                        // AArch64 only
          (HMC:SSC != '000' && HMC:SSC != '111' && !HaveEL(EL3) && !HaveEL(EL2)) || // No EL3/EL2
          (HMC:SSC:PxC == '11100' && !HaveEL(EL2))) then                            // No EL2
        (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    PL2_match = HaveEL(EL2) && HMC == '1';
    PL1_match = PxC<0> == '1';
    PL0_match = PxC<1> == '1';
    SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';

    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 = TRUE;         // Both

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));
        last_ctx_cmp = UInt(DBGDIDR.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCT
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(32) UNKNOWN;
        linked_to = TRUE;
        (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

```
// AArch32.GenerateDebugExceptions()
// =================================

boolean AArch32.GenerateDebugExceptions()
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());
```

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```
// AArch32.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)

    if from == EL0 && !ELStateUsingAArch32(EL1, secure) then
        mask = bit UNKNOWN;                          // PSTATE.D mask, unused for EL0 case
        return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    if HaveEL(EL3) && secure then
        spd = (if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32);
        if spd<1> == '1' then
            enabled = spd<0> == '1';
        else
            // SPD == 0b01 is reserved, but behaves the same as 0b00.
            enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from != EL2;

    return enabled;
```

Library pseudocode for aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```
// AArch32.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch32.CheckForPMUOverflow()

    if !ELUsingAArch32(EL1) then return AArch64.CheckForPMUOverflow();
    pmuirq = (PMCR.E == '1' && PMINTENSET<31> == '1' && PMOVSSET<31> == '1');
    for n = 0 to UInt(PMCR.N) - 1
        if HaveEL(EL2) then
            hpmn = (if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN);
            hpme = (if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME);
            E = (if n < UInt(hpmn) then PMCR.E else hpme);
        else
            E = PMCR.E;
        if E == '1' && PMINTENSET<n> == '1' && PMOVSSET<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

    return pmuirq;
```

Library pseudocode for aarch32/debug/pmu/AArch32.CountEvents

```
// AArch32.CountEvents()
// =====================
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch32.CountEvents(integer n)
    assert (n == 31 || n < UInt(PMCR.N));

    if !ELUsingAArch32(EL1) then return AArch64.CountEvents(n);
    // Event counting is disabled in Debug state
    debug = Halted();

    // In Non-secure state, some counters are reserved for EL2
    if HaveEL(EL2) then
        hpmn = (if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN);
        hpme = (if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME);
        E = (if n < UInt(hpmn) || n == 31 then PMCR.E else hpme);
    else
        E = PMCR.E;
    enabled = (E == '1' && PMCNTENSET<n> == '1');

    if !IsSecure() then
        // Event counting in Non-secure state is allowed unless all of:
        // * EL2 and the HPMD Extension are implemented
        // * Executing at EL2
        // * PMNx is not reserved for EL2
        // * HDCR.HPMD == 1
        if HaveHPMDExt() && PSTATE.EL == EL2 && (n < UInt(hpmn) || n == 31) then
            hpmd = (if !ELUsingAArch32(EL2) then MDCR_EL2.HPMD else HDCR.HPMD);
            prohibited = (hpmd == '1');
        else
            prohibited = FALSE;
    else
        // Event counting in Secure state is prohibited unless any one of:
        // * EL3 is not implemented
        // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
        // * EL3 is using AArch32 and SDCR.SPME == 1
        // * Executing at EL0, and SDER.SUNIDEN == 1.
        spme = (if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME);
        prohibited = (HaveEL(EL3) && spme == '0' && (PSTATE.EL != EL0 || SDER.SUNIDEN == '0'));

    // The IMPLEMENTATION DEFINED authentication interface might override software controls
    if ExternalSecureNoninvasiveDebugEnabled() then prohibited = FALSE;

    // For the cycle counter, PMCR.DP enables counting when otherwise prohibited
    if prohibited && n == 31 then prohibited = (PMCR.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
    filter = (if n == 31 then PMCCFILTR else PMEVTYPER[n]);

    P = filter<31>;
    U = filter<30>;
    NSK = (if HaveEL(EL3) then filter<29> else '0');
    NSU = (if HaveEL(EL3) then filter<28> else '0');
    NSH = (if HaveEL(EL2) then filter<27> else '0');

    case PSTATE.EL of
        when EL0  filtered = (if IsSecure() then U == '1' else U != NSU);
        when EL1  filtered = (if IsSecure() then P == '1' else P != NSK);
        when EL2  filtered = (NSH == '0');
        when EL3  filtered = (P == '1');

    return !debug && enabled && !prohibited && !filtered;
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// ==================================
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    EDSCR.ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction();
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// ===============================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then
        PSTATE.PAN = '1';
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    EndOfInstruction();
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// ======================================
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    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() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    EndOfInstruction();
```

Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =============================

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)

    bottom = if DBGWVR[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));  MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                     // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                                        // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;            // Disabled
            when Constraint_NONE      mask = 0;                // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<31:mask> == DBGWVR[n]<31:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
        WVR_match = vaddress<31:bottom> == DBGWVR[n]<31:bottom>;

    return WVR_match && byte_select_match;
```

Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
                                boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIDR.WRPs);

    // "ispriv" is FALSE for LDRT/STRT instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, DBGWCR[n].LBN, isbreakpnt, ispriv);

    ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                            (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress), fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort  exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception type, FaultRecord fault, bits(32) vaddress)

    exception = ExceptionSyndrome(type);

    d_side = type == Exception_DataAbort;

    exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.ipaddress = ZeroExtend(fault.ipaddress);
    else
        exception.ipavalid = FALSE;

    return exception;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// ==========================

AArch32.CheckPCAlignment()

    bits(32) pc = ThisInstrAddr();
    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

        // Generate an Alignment fault Prefetch Abort exception
        vaddress = pc;
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        secondstage = FALSE;
        AArch32.Abort(vaddress, AArch32.AlignmentFault(acctype, iswrite, secondstage));
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

    // The encoding used in the IFSR or DFSR can be Long-descriptor format or Short-descriptor
    // format.  Normally, the current translation table format determines the format. For an abort
    // from Non-secure state to Monitor mode, the IFSR or DFSR uses the Long-descriptor format if
    // any of the following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * The abort is synchronous and either:
    //   - It is taken from Hyp mode.
    //   - It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1';
        if !IsSErrorInterrupt(fault) && !long_format then
            long_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';
    d_side = TRUE;
    if long_format then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType_IC then
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// =============================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';

    d_side = FALSE;
    if long_format then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// ================================

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || IsSecondStage(fault) ||
                    (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                    (IsDebugException(fault) && HDCR.TDE == '1')));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = 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);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// ====================================

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || IsSecondStage(fault) ||
                    (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                    (IsDebugException(fault) && HDCR.TDE == '1')));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0C;
    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        if fault.type == Fault_Alignment then             // PC Alignment fault
            exception = ExceptionSyndrome(Exception_PCAlignment);
            exception.vaddress = ThisInstrAddr();
        else
            exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/asynch/AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// ==================================

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !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 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || HCR.FMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_FIQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/asynch/AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !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 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_IRQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/asynch/AArch32.TakePhysicalSErrorException

```
// AArch32.TakePhysicalSErrorException()
// =====================================

AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) errortype,
                                    boolean impdef_syndrome, bits(24) full_syndrome)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSErrorException(impdef_syndrome, full_syndrome);

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || HCR.AMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);
    vaddress = bits(32) UNKNOWN;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(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);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// =================================

AArch32.TakeVirtualFIQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    if ELUsingAArch32(EL2) then  // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR.TGE == '0' && HCR.FMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// =================================

AArch32.TakeVirtualIRQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};

    if ELUsingAArch32(EL2) then   // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualSErrorException

```
// AArch32.TakeVirtualSErrorException()
// ====================================

AArch32.TakeVirtualSErrorException(bit extflag, bits(2) errortype, boolean impdef_syndrome, bits(24) fu

    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    if ELUsingAArch32(EL2) then   // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualSErrorException(impdef_syndrome

    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);
    if ELUsingAArch32(EL2) then HCR.VA = '0'; else HCR_EL2.VSE = '0';
    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 (HaveEL(EL2) && !IsSecure() && !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.ExceptionClass

```
// AArch32.ExceptionClass()
// ========================
// Return the Exception Class and Instruction Length fields for reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception type)

    il = if ThisInstrLength() == 32 then '1' else '0';

    case type of
        when Exception_Uncategorized        ec = 0x00; il = '1';
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03;
        when Exception_CP15RRTTrap          ec = 0x04;
        when Exception_CP14RTTrap           ec = 0x05;
        when Exception_CP14DTTrap           ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap  ec = 0x07;
        when Exception_FPIDTrap             ec = 0x08;
        when Exception_CP14RRTTrap          ec = 0x0C;
        when Exception_IllegalState         ec = 0x0E; il = '1';
        when Exception_SupervisorCall       ec = 0x11;
        when Exception_HypervisorCall       ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_InstructionAbort     ec = 0x20; il = '1';
        when Exception_PCAlignment          ec = 0x22; il = '1';
        when Exception_DataAbort            ec = 0x24;
        when Exception_FPTrappedException   ec = 0x28;
        otherwise                           Unreachable();

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;

    return (ec,il);
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

```
// AArch32.GeneralExceptionsToAArch64()
// ====================================
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception type = exception.type;

    (ec,il) = AArch32.ExceptionClass(type);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if type IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif type == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch32.ResetControlRegisters(boolean cold_reset);
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// ===================
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                       // Secure state
    elsif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset the CP14 and CP15 registers and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111';       // All asynchronous exceptions masked
    PSTATE.IT = '00000000';       // IT block state reset
    PSTATE.T = SCTLR.TE;          // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
    PSTATE.E = SCTLR.EE;          // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0';              // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv;                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        if MVBAR<0> == '1' then           // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';
    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    BranchTo(rv, BranchType_UNKNOWN);
```

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, element, accumulated_exceptions);
    FPEXC.DEX    = '1';
    FPEXC.TFV    = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF

    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.TakeHVCException(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.TakeSVCException(immediate);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCException

```
// AArch32.TakeHVCException()
// ==========================

AArch32.TakeHVCException(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 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_SupervisorCall);
        exception.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// ======================
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return,
                     integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    spsr = GetPSRFromPSTATE();
    if !(exception.type IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_UNKNOWN);

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// ===================
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                  integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    spsr = GetPSRFromPSTATE();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                         // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then
        PSTATE.PAN = '1';
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_UNKNOWN);

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// ==========================
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                         integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    spsr = GetPSRFromPSTATE();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    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';
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_UNKNOWN);

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.AArch32SystemAccessTrap

```
// AArch32.AArch32SystemAccessTrap()
// =================================
// Trapped AArch32 System register access other than due to CPTR_EL2 or CPACR_EL1.

AArch32.AArch32SystemAccessTrap(bits(2) target_el, bits(32) instr)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if !ELUsingAArch32(target_el) || AArch32.GeneralExceptionsToAArch64() then
        AArch64.AArch32SystemAccessTrap(target_el, instr);

    assert target_el IN {EL1,EL2};

    if target_el == EL2 then
        exception = AArch32.AArch32SystemAccessTrapSyndrome(instr);
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.AArch32SystemAccessTrapSyndrome

```
// AArch32.AArch32SystemAccessTrapSyndrome()
// =========================================
// Return the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.AArch32SystemAccessTrapSyndrome(bits(32) instr)

    ExceptionRecord exception;
    cpnum = UInt(instr<11:8>);

    bits(20) iss = Zeros();
    if instr<27:24> == '1110' && instr<4> == '1' && instr<31:28> != '1111' then
        // MRC/MCR
        case cpnum of
            when 10    exception = ExceptionSyndrome(Exception_FPIDTrap);
            when 14    exception = ExceptionSyndrome(Exception_CP14RTTrap);
            when 15    exception = ExceptionSyndrome(Exception_CP15RTTrap);
            otherwise  Unreachable();
        iss<19:17> = instr<7:5>;      // opc2
        iss<16:14> = instr<23:21>;    // opc1
        iss<13:10> = instr<19:16>;    // CRn
        iss<8:5>   = instr<15:12>;    // Rt
        iss<4:1>   = instr<3:0>;      // CRm
    elsif instr<27:21> == '1100010' && instr<31:28> != '1111' then
        // MRRC/MCRR
        case cpnum of
            when 14    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
            when 15    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
            otherwise  Unreachable();
        iss<19:16> = instr<7:4>;      // opc1
        iss<13:10> = instr<19:16>;    // Rt2
        iss<8:5>   = instr<15:12>;    // Rt
        iss<4:1>   = instr<3:0>;      // CRm
    elsif instr<27:25> == '110' && instr<31:28> != '1111' then
        // LDC/STC
        assert cpnum == 14;
        exception = ExceptionSyndrome(Exception_CP14DTTrap);
        iss<19:12> = instr<7:0>;      // imm8
        iss<4>     = instr<23>;       // U
        iss<2:1>   = instr<24,21>;    // P,W
        if instr<19:16> == '1111' then    // Rn==15, LDC(Literal addressing)/STC
            iss<8:5> = bits(4) UNKNOWN;
            iss<3>   = '1';
        else
            iss<8:5> = instr<19:16>;  // Rn
            iss<3>   = '0';
    else
        Unreachable();
    iss<0> = instr<20>;               // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0>  = iss;

    return exception;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEnabled

```
// AArch32.CheckAdvSIMDOrFPEnabled()
// =================================
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)
    if PSTATE.EL == EL0 && (!HaveEL(EL2) || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0')) && !ELUsingAA
        // The PE behaves as if FPEXC.EN is 1
        AArch64.CheckFPAdvSIMDEnabled();
    elsif PSTATE.EL == EL0 && HaveEL(EL2) && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        if fpexc_check && HCR_EL2.RW == '0' then
            fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
            if fpexc_en == '0' then UNDEFINED;
        AArch64.CheckFPAdvSIMDEnabled();
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then UNDEFINED;

            if cpacr_cp10 == '10' then
                (c, cpacr_cp10) = ConstrainUnpredictableBits(Unpredictable_RESCPACR);

            // Check if access disabled in CPACR
            case cpacr_cp10 of
                when '00'  disabled = TRUE;
                when '01'  disabled = PSTATE.EL == EL0;
                when '11'  disabled = FALSE;
            if disabled then UNDEFINED;

        // If required, check FPEXC enabled bit.
        if fpexc_check && FPEXC.EN == '0' then UNDEFINED;

        AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if HaveEL(EL2) && !IsSecure() && !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 HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCUndefOrTrap(Zeros(16));
    else
        route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            exception = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(exception);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWFxTrap

```
// AArch32.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        AArch64.CheckForWFxTrap(target_el, is_wfe);
        return;
    case target_el of
        when EL1 trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2 trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3 trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';
    if trap then
        if (target_el == EL1 && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) &&
            HCR_EL2.TGE == '1') then
            AArch64.WFxTrap(target_el, is_wfe);
        if target_el == EL3 then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            exception = ExceptionSyndrome(Exception_WFxTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
            exception.syndrome<0> = if is_wfe then '1' else '0';
            AArch32.TakeHypTrapException(exception);
        else
            AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)

    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

        // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
        next_instr = AArch32.MemSingle[NextInstrAddr(), 2, AccType_IFETCH, TRUE];

        if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx',
                          '01001xxxxxxxxxxx', '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 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// ============================
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()

    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[].SED);
    if setend_disabled == '1' then
        UNDEFINED;

    return;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x14;

    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// ==================================
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException()
    exception = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException(ExceptionRecord exception)

    route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR.TGE == '1';

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    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.CreateFaultRecord

```
// AArch32.CreateFaultRecord()
// ===========================

FaultRecord AArch32.CreateFaultRecord(Fault type, bits(40) ipaddress,  bits(4) domain,
                                      integer level, AccType acctype, boolean write, bit extflag,
                                      bits(4) debugmoe, bits(2) errortype, boolean secondstage, boolean

    FaultRecord fault;
    fault.type = type;
    if (type != Fault_None && PSTATE.EL != EL2 && TTBCR.EAE == '0' && !secondstage && !s2fs1walk &&
        AArch32.DomainValid(type, level)) then
        fault.domain = domain;
    else
        fault.domain = bits(4) UNKNOWN;
    fault.debugmoe = debugmoe;
    fault.errortype = errortype;
    fault.ipaddress = ZeroExtend(ipaddress);
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fs1walk = s2fs1walk;

    return fault;
```

Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid

```
// AArch32.DomainValid()
// =====================
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault type, integer level)
    assert type != Fault_None;

    case type of
        when Fault_Domain
            return TRUE;
        when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusLD

```
// AArch32.FaultStatusLD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '1';
    fsr<5:0> = EncodeLDFSC(fault.type, fault.level);

    return fsr;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusSD

```
// AArch32.FaultStatusSD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '0';
    fsr<10,3:0> = EncodeSDFSC(fault.type, fault.level);
    if d_side then
        fsr<7:4> = fault.domain;               // Domain field (data fault only)

    return fsr;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

```
// AArch32.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;

    bits(25) iss = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then iss<11:10> = fault.errortype; // AET
    if d_side then
        if IsSecondStage(fault) && !fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.type, fault.level);

    return iss;
```

Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault type, integer level)

    bits(5) result;
    case type of
        when Fault_AccessFlag
            assert level IN {1,2};
            result = if level == 1 then '00011' else '00110';
        when Fault_Alignment
            result = '00001';
        when Fault_Permission
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault_Domain
            assert level IN {1,2};
            result = if level == 1 then '01001' else '01011';
        when Fault_Translation
            assert level IN {1,2};
            result = if level == 1 then '00101' else '00111';
        when Fault_SyncExternal
            result = '01000';
        when Fault_SyncExternalOnWalk
            assert level IN {1,2};
            result = if level == 1 then '01100' else '01110';
        when Fault_SyncParity
            result = '11001';
        when Fault_SyncParityOnWalk
            assert level IN {1,2};
            result = if level == 1 then '11100' else '11110';
        when Fault_AsyncParity
            result = '11000';
        when Fault_AsyncExternal
            result = '10110';
        when Fault_Debug
            result = '00010';
        when Fault_TLBConflict
            result = '10000';
        when Fault_Lockdown
            result = '10100';   // IMPLEMENTATION DEFINED
        when Fault_Exclusive
            result = '10101';   // IMPLEMENTATION DEFINED
        when Fault_ICacheMaint
            result = '00100';
        otherwise
            Unreachable();

    return result;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm_C

```
// A32ExpandImm_C()
// ================

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRType_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

Library pseudocode for aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

Library pseudocode for aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) type)
    case type of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;
```

Library pseudocode for aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

Library pseudocode for aarch32/functions/common/RRX_C

```
// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

Library pseudocode for aarch32/functions/common/SRType

```
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

Library pseudocode for aarch32/functions/common/Shift

```
// Shift()
// =======

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, type, amount, carry_in);
    return result;
```

Library pseudocode for aarch32/functions/common/Shift_C

```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```

Library pseudocode for aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// ==============

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

Library pseudocode for aarch32/functions/common/T32ExpandImm_C

```
// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained CP15 traps in HSTR and HCR.

boolean AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        if PSTATE.EL == EL0 && !ELUsingAArch32(EL2) then
            return AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !(major IN {4,14}) && HSTR<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR.TIDCP
        if (HCR.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
             (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```

Library pseudocode for aarch32/functions/coproc/AArch32.CheckSystemAccess

```
// AArch32.CheckSystemAccess()
// ===========================
// Check System register access instruction for enables and disables

AArch32.CheckSystemAccess(integer cp_num, bits(32) instr)
    assert cp_num == UInt(instr<11:8>) && (cp_num IN {14,15});

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        AArch64.CheckAArch32SystemAccess(instr);
        return;

    // Decode the AArch32 System register access instruction
    if instr<31:28> != '1111' && instr<27:24> == '1110' && instr<4> == '1' then      // MRC/MCR
        cprt = TRUE;  cpdt = FALSE;  nreg = 1;
        opc1 = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn  = UInt(instr<19:16>);
        CRm  = UInt(instr<3:0>);
    elsif instr<31:28> != '1111' && instr<27:21> == '1100010' then                   // MRRC/MCRR
        cprt = TRUE;  cpdt = FALSE;  nreg = 2;
        opc1 = UInt(instr<7:4>);
        CRm  = UInt(instr<3:0>);
    elsif instr<31:28> != '1111' && instr<27:25> == '110' && instr<22> == '0' then   // LDC/STC
        cprt = FALSE;  cpdt = TRUE;  nreg = 0;
        opc1 = 0;
        CRn  = UInt(instr<15:12>);
    else
        allocated = FALSE;

    //
    // Coarse-grain decode into CP14 or CP15 encoding space. Each of the CPxxxInstrDecode functions
    // returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
    if cp_num == 14 then
        // LDC and STC only supported for c5 in CP14 encoding space
        if cpdt && CRn != 5 then
            allocated = FALSE;
        else
            // Coarse-grained decode of CP14 based on opc1 field
            case opc1 of
                when 0     allocated = CP14DebugInstrDecode(instr);
                when 1     allocated = CP14TraceInstrDecode(instr);
                when 7     allocated = CP14JazelleInstrDecode(instr);    // JIDR only
                otherwise  allocated = FALSE;          // All other values are unallocated

    elsif cp_num == 15 then
        // LDC and STC not supported in CP15 encoding space
        if !cprt then
            allocated = FALSE;
        else
            allocated = CP15InstrDecode(instr);

            // Coarse-grain traps to EL2 have a higher priority than exceptions generated because
            // the access instruction is UNDEFINED
            if AArch32.CheckCP15InstrCoarseTraps(CRn, nreg, CRm) then
                // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
                // Non-secure User mode is UNDEFINED when the trap is disabled, then it is
                // IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
                // when the trap is enabled.
                if PSTATE.EL == EL0 && !IsSecure() && !allocated then
                    if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at NS EL0" then
                        UNDEFINED;
                AArch32.AArch32SystemAccessTrap(EL2, instr);

    else
        allocated = FALSE;

    if !allocated then
        UNDEFINED;

    // If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
```

```
    AArch32.CheckSystemAccessTraps(instr);

    return;
```

Library pseudocode for aarch32/functions/coproc/AArch32.CheckSystemAccessEL1Traps

```
// AArch32.CheckSystemAccessEL1Traps()
// ===================================
// Check for configurable disables or traps to EL1 or EL2 of a System register
// access instruction.

AArch32.CheckSystemAccessEL1Traps(bits(32) instr)
    assert PSTATE.EL == EL0;

    if ((HaveEL(EL1) && IsSecure() && !ELUsingAArch32(EL1)) || IsInHost()) then
        AArch64.CheckAArch32SystemAccessEL1Traps(instr);
        return;
    trap = FALSE;

    // Decode the AArch32 System register access instruction
    (op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

    if cp_num == 14 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 == 0) || // DBGDTRR
            (op == SystemAccessType_DT && CRn == 5 && opc2 == 0)) then                      // DBGDTRR
            trap = !Halted() && DBGDSCRext.UDCCdis == '1';

        elsif opc1 == 0 then
            trap = DBGDSCRext.UDCCdis == '1';

        elsif opc1 == 1 then
            trap = CPACR.TRCDIS == '1';
            if HaveEL(EL3) && ELUsingAArch32(EL3) && NSACR.NSTRCDIS == '1' then
                trap = TRUE;

    elsif cp_num == 15 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0) || // PMCR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 1) || // PMCNTE
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 2) || // PMCNTE
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 3) || // PMOVSR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 6) || // PMCEID
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 7) || // PMCEID
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 1) || // PMXEVT
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 14 && opc2 == 3) || // PMOVSS
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 12)) then          // PMEVTY
            trap = PMUSERENR.EN == '0';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 14 && opc2 == 4 then // PMSW
            trap = PMUSERENR.EN == '0' && PMUSERENR.SW == '0';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 0) || // PMC
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 9)) then                         // PMC
            trap = PMUSERENR.EN == '0' && (write || PMUSERENR.CR == '0');

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 2) || // PMX
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8 && CRm <= 11)) then  // PM
            trap = PMUSERENR.EN == '0' && (write || PMUSERENR.ER == '0');

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 5 then    // P
            trap = PMUSERENR.EN == '0' && PMUSERENR.ER == '0';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then /
            trap = CNTKCTL.PL0PTEN == '0';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 0 && opc2 == 0 then       /
            trap = CNTKCTL.PL0PCTEN == '0' && CNTKCTL.PL0VCTEN == '0';

        elsif op == SystemAccessType_RRT && opc1 == 1 && CRm == 14 then                               /
            trap = CNTKCTL.PL0VCTEN == '0';

    if trap then
        AArch32.AArch32SystemAccessTrap(EL1, instr);
```

Library pseudocode for aarch32/functions/coproc/AArch32.CheckSystemAccessEL2Traps

```
// AArch32.CheckSystemAccessEL2Traps()
// ===================================
// Check for configurable traps to EL2 of a System register access instruction.

AArch32.CheckSystemAccessEL2Traps(bits(32) instr)
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1};

    if HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        AArch64.CheckAArch32SystemAccessEL2Traps(instr);
        return;
    trap = FALSE;

    // Decode the AArch32 System register access instruction
    (op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

    if cp_num == 14 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 0) || // DBGDRAR
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 1) ||                         // DBGDRAR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 0) || // DBGDSAR
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 2)) then                      // DBGDSAR
            trap = HDCR.TDRA == '1' || HDCR.TDE == '1' || HCR.TGE == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 4) || // DBGO
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 4) ||    // DBGO
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 4) ||    // DBGO
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 4 && opc2 == 4)) then // DBGP
            trap = HDCR.TDOSA == '1' || HDCR.TDE == '1' || HCR.TGE == '1';

        elsif opc1 == 0 && (!Halted() || !(op == SystemAccessType_RT && CRn == 0 && CRm == 5 && opc2 ==
            trap = HDCR.TDA == '1' || HDCR.TDE == '1' || HCR.TGE == '1';

        elsif opc1 == 1 then
            trap = HCPTR.TTA == '1';
            if HaveEL(EL3) && ELUsingAArch32(EL3) && NSACR.NSTRCDIS == '1' then
                trap = TRUE;

        elsif op == SystemAccessType_RT && opc1 == 7 && CRn == 0 && CRm == 0 && opc2 == 0 then  // JIDR
            trap = HCR.TID0 == '1';

    elsif cp_num == 15 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 0) ||  // SCTLR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 0) ||  // TTBR0
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 2) ||                          // TTBR0
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 1) ||  // TTBR1
            (op == SystemAccessType_RRT && opc1 == 1 && CRm == 2) ||                          // TTBR1
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 2) ||  // TTBCR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 3 && CRm == 0 && opc2 == 0) ||  // DACR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 5 && CRm == 0 && opc2 == 0) ||  // DFSR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 5 && CRm == 0 && opc2 == 1) ||  // IFSR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 6 && CRm == 0 && opc2 == 0) ||  // DFAR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 6 && CRm == 0 && opc2 == 2) ||  // IFAR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 5 && CRm == 1 && opc2 == 0) ||  // ADFSR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 5 && CRm == 1 && opc2 == 1) ||  // AIFSR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 10 && CRm == 2 && opc2 == 0) || // PRRR/M
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 10 && CRm == 2 && opc2 == 1) || // NMRR/M
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 10 && CRm == 3 && opc2 == 0) || // AMAIR0
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 10 && CRm == 3 && opc2 == 1) || // AMAIR1
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 13 && CRm == 0 && opc2 == 1)) then // CON
            trap = if write then HCR.TVM == '1' else HCR.TRVM == '1';
        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 8 then                         // TLBI
            trap = write && HCR.TTLB == '1';
        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 6 && opc2 == 2) ||  // DCI
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 10 && opc2 == 2) ||    // DCC
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 14 && opc2 == 2)) then // DCC
            trap = write && HCR.TSW == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 6 && opc2 == 1) ||  // DCI
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 10 && opc2 == 1) ||    // DCC
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 14 && opc2 == 1)) then // DCC
            trap = write && HCR.TPC == '1';
```

```
        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 1) ||  // ICI
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 0) ||     // ICI
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 1 && opc2 == 0) ||     // ICI
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 11 && opc2 == 1)) then // DCC
            trap = write && HCR.TPU == '1';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 1 then   // ACT
            trap = HCR.TAC == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 2) ||  // TCM
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 3) ||     // TLB
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 6) ||     // REV
            (op == SystemAccessType_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 7)) then  // AID
            trap = HCR.TID1 == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 1) ||  // CTR
            (op == SystemAccessType_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 0) ||     // CCS
            (op == SystemAccessType_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 1) ||     // CLI
            (op == SystemAccessType_RT && opc1 == 2 && CRn == 0 && CRm == 0 && opc2 == 0)) then  // CSS
            trap = HCR.TID2 == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 1) ||               // ID_
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 2 && opc2 <= 5) ||     // ID_
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm >= 3 && opc2 <= 1) ||     // Res
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 3 && opc2 == 2) ||     // Res
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 IN {4,5})) then  //
            trap = HCR.TID3 == '1';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 2 then   // CPA
            trap = HCPTR.TCPAC == '1';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0 then  // PMC
            trap = HDCR.TPMCR == '1' || HDCR.TPM == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8) ||      // PMEVCNTR<n>
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm IN {12,13,14}) || // PM*
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 9)) then                  // PMCCNTR (MR
            trap = HDCR.TPM == '1';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then
            trap = CNTHCTL.PL1PCEN == '0';
        elsif op == SystemAccessType_RRT && opc1 == 0 && CRm == 14 then
            trap = CNTHCTL.PL1PCTEN == '0';

    if trap then
        AArch32.AArch32SystemAccessTrap(EL2, instr);
```

Library pseudocode for aarch32/functions/coproc/AArch32.CheckSystemAccessTraps

```
// AArch32.CheckSystemAccessTraps()
// ================================
// Check for configurable disables or traps to a higher EL of an System register access.

AArch32.CheckSystemAccessTraps(bits(32) instr)

    if PSTATE.EL == EL0 then
        AArch32.CheckSystemAccessEL1Traps(instr);
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        AArch32.CheckSystemAccessEL2Traps(instr);
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && PSTATE.EL IN {EL0,EL1,EL2} then
        AArch64.CheckAArch32SystemAccessEL3Traps(instr);
```

Library pseudocode for aarch32/functions/coproc/AArch32.DecodeSysRegAccess

```
// AArch32.DecodeSysRegAccess()
// ============================
// Decode an AArch32 System register access instruction into its operands.

(SystemAccessType, integer, integer, integer, integer, integer, boolean) AArch32.DecodeSysRegAccess(bit

    cp_num = UInt(instr<11:8>);

    // Decode the AArch32 System register access instruction
    if instr<31:28> != '1111' && instr<27:24> == '1110' && instr<4> == '1' then   // MRC/MCR
        op    = SystemAccessType_RT;
        opc1  = UInt(instr<23:21>);
        opc2  = UInt(instr<7:5>);
        CRn   = UInt(instr<19:16>);
        CRm   = UInt(instr<3:0>);
        write = instr<20> == '0';
    elsif instr<31:28> != '1111' && instr<27:21> == '1100010' then               // MRRC/MCRR
        op    = SystemAccessType_RRT;
        opc1  = UInt(instr<7:4>);
        CRm   = UInt(instr<3:0>);
        write = instr<20> == '0';
    elsif instr<31:28> != '1111' && instr<27:25> == '110' then                   // LDC/STC
        op    = SystemAccessType_DT;
        CRn   = UInt(instr<15:12>);
        write = instr<20> == '0';

    return (op, cp_num, opc1, CRn, CRm, opc2, write);
```

Library pseudocode for aarch32/functions/coproc/CP14DebugInstrDecode

```
// Decodes an accepted access to a debug System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14DebugInstrDecode(bits(32) instr);
```

Library pseudocode for aarch32/functions/coproc/CP14JazelleInstrDecode

```
// Decodes an accepted access to a Jazelle System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14JazelleInstrDecode(bits(32) instr);
```

Library pseudocode for aarch32/functions/coproc/CP14TraceInstrDecode

```
// Decodes an accepted access to a trace System register in the CP14 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP14TraceInstrDecode(bits(32) instr);
```

Library pseudocode for aarch32/functions/coproc/CP15InstrDecode

```
// Decodes an accepted access to a System register in the CP15 encoding space.
// Returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
boolean CP15InstrDecode(bits(32) instr);
```

Library pseudocode for aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// ===============================

// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

Library pseudocode for aarch32/functions/exclusive/AArch32.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

Library pseudocode for aarch32/functions/exclusive/AArch32.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```

Library pseudocode for aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// ==============================

// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = (address != Align(address, size));
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====================

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```

Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

```
// CheckAdvSIMDOrVFPEnabled()
// ==========================

CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// ======================

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/CheckVFPEnabled

```
// CheckVFPEnabled()
// =================

CheckVFPEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() 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, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 != 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};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0');
        result = FPHalvedSub(three, product, fpcr);
    return result;
```

Library pseudocode for aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =============

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == 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) two = FPTwo('0');
        result = FPSub(two, product, fpcr);
    return result;
```

Library pseudocode for aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// ====================

FPCRType StandardFPSCRValue()
    return '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '0000000000000000000';
```

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 };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED };
    vector  = acctype == AccType_VEC;

    // AccType_VEC is used for SIMD element alignment checks only
    check = (atomic || ordered || vector || A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

Library pseudocode for aarch32/functions/memory/AArch32.MemSingle

```
// AArch32.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    value = _Mem[memaddrdesc, size, accdesc];
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8) v
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    _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/MemU

```
// MemU[] - non-assignment form
// ============================

bits(8*size) MemU[bits(32) address, integer size]
    acctype = AccType_NORMAL;
    return Mem_with_type[address, size, acctype];

// MemU[] - assignment form
// ========================

MemU[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_NORMAL;
    Mem_with_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemU_unpriv

```
// MemU_unpriv[] - non-assignment form
// ===================================

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType_UNPRIV;
    return Mem_with_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// ===============================

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_UNPRIV;
    Mem_with_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/Mem_with_type

```
// Mem_with_type[] - non-assignment (read) form
// ============================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    integer i;
    boolean iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
    if !aligned then
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned];

    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem_with_type[] - assignment (write) form
// =========================================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
    integer i;
    boolean iswrite = TRUE;

    if BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    else
        AArch32.MemSingle[address, size, acctype, aligned] = value;
    return;
```

Library pseudocode for aarch32/functions/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 && HaveEL(EL2) && !IsSecure() && !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 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR.TGE == '1' || HCR.AMO == '1'));

    if route_to_monitor then
        target = M32_Monitor;
    elsif route_to_hyp || PSTATE.M == M32_Hyp then
        target = M32_Hyp;
    else
        target = M32_Abort;

    if IsSecure() then
        mask_active = TRUE;
    elsif target == M32_Monitor then
        mask_active = (SCR.AW == '1' &&
                       (!HaveEL(EL2) || (HCR.TGE == '0' && HCR.AMO == '0')));
    else
        mask_active = (target == M32_Abort || PSTATE.M == M32_Hyp);

    mask_set = (PSTATE.A == '1');
    (-, el)  = ELFromM32(target);
    intdis   = (Halted() || ExternalDebugInterruptsDisabled(el));
    masked   = intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending
    if SErrorPending() && masked then          // i.e. ISR.A is set to 1
        syndrome32 = AArch32.PhysicalSErrorSyndrome();
        DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        ClearPendingPhysicalSError();                  // Clear ISR.A to 0

    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 HaveAnyAArch64() then
        target<24:0> = ZeroExtend(syndrome);// Any RES0 fields must be set to zero
    else
        target<15:0> = syndrome;
    return target;
```

Library pseudocode for aarch32/functions/ras/AArch32.SErrorSyndrome

```
type AArch32.SErrorSyndrome is (
    bits(2) AET,
    bit ExT
)
```

Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

```
// AArch32.vESBOperation()
// =======================
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESBOperation();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a
    // virtual SError interrupt might be pending
    vSEI_enabled = (HCR.TGE == '0' && HCR.AMO == '1');
    vSEI_pending = (vSEI_enabled && HCR.VA == '1');
    vintdis      = (Halted() || ExternalDebugInterruptsDisabled(EL1));
    vmasked      = (vintdis || PSTATE.A == '1');

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        VDISR = AArch32.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 = bits(32) UNKNOWN;
    SPSR_irq = bits(32) UNKNOWN;
    SPSR_svc = bits(32) UNKNOWN;
    SPSR_abt = bits(32) UNKNOWN;
    SPSR_und = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

```
AArch32.ResetSystemRegisters(boolean cold_reset);
```

Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

```
// ALUExceptionReturn()
// ====================

ALUExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;          // UNDEFINED or NOP
    else
        AArch32.ExceptionReturn(address, SPSR[]);
```

Library pseudocode for aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address);
    else
        BranchWritePC(address);
```

Library pseudocode for aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address)
    if address<0> == '1' then
        SelectInstrSet(InstrSet_T32);
        address<0> = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the processor takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address<1> == '1' && ConstrainUnpredictableBool(Unpredictable_A32FORCEALIGNPC) then
            address<1> = '0';
    BranchTo(address, BranchType_UNKNOWN);
```

Library pseudocode for aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        address<1:0> = '00';
    else
        address<0> = '0';
    BranchTo(address, BranchType_UNKNOWN);
```

Library pseudocode for aarch32/functions/registers/D

```
// D[] - non-assignment form
// =========================

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    return _V[n DIV 2]<base+63:base>;

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    _V[n DIV 2]<base+63:base> = value;
    return;
```

Library pseudocode for aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];
```

Library pseudocode for aarch32/functions/registers/LR

```
// LR - assignment form
// ====================

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// ========================

bits(32) LR
    return R[14];
```

Library pseudocode for aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address);
```

Library pseudocode for aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// ==============

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    case n of  // Select  index by mode:     usr fiq irq svc abt und hyp
        when 8     result = RBankSelect(mode,  8, 24,  8,  8,  8,  8,  8);
        when 9     result = RBankSelect(mode,  9, 25,  9,  9,  9,  9,  9);
        when 10    result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11    result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12    result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13    result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14    result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise  result = n;

    return result;
```

Library pseudocode for aarch32/functions/registers/Monitor_mode_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

Library pseudocode for aarch32/functions/registers/PC

```
// PC - non-assignment form
// ========================

bits(32) PC
    return R[15];               // This includes the offset from AArch32 state
```

Library pseudocode for aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// ==============

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before ARMv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

Library pseudocode for aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =========================

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return _V[n];

// Q[] - assignment form
// =====================

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    _V[n] = value;
    return;
```

Library pseudocode for aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// ===========================

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

Library pseudocode for aarch32/functions/registers/R

```
// R[] - assignment form
// =====================

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =========================

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];
```

Library pseudocode for aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                    integer svc, integer abt, integer und, integer hyp)

    case mode of
        when M32_User    result = usr;  // User mode
        when M32_FIQ     result = fiq;  // FIQ mode
        when M32_IRQ     result = irq;  // IRQ mode
        when M32_Svc     result = svc;  // Supervisor mode
        when M32_Abort   result = abt;  // Abort mode
        when M32_Hyp     result = hyp;  // Hyp mode
        when M32_Undef   result = und;  // Undefined mode
        when M32_System  result = usr;  // System mode uses User mode registers
        otherwise        Unreachable(); // Monitor mode

    return result;
```

Library pseudocode for aarch32/functions/registers/Rmode

```
// Rmode[] - non-assignment form
// ============================

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =========================

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if !HighestELUsingAArch32() && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[LookUpRIndex(n, mode)] = ZeroExtend(value);
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;

    return;
```

Library pseudocode for aarch32/functions/registers/S

```
// S[] - non-assignment form
// =========================

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    return _V[n DIV 4]<base+31:base>;

// S[] - assignment form
// =====================

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    _V[n DIV 4]<base+31:base> = value;
    return;
```

Library pseudocode for aarch32/functions/registers/SP

```
// SP - assignment form
// ====================

SP = bits(32) value
    R[13] = value;
    return;

// SP - non-assignment form
// ========================

bits(32) SP
    return R[13];
```

Library pseudocode for aarch32/functions/registers/_Dclone

```
array bits(64) _Dclone[0..31];
```

Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =========================

AArch32.ExceptionReturn(bits(32) new_pc, bits(32) spsr)

    SynchronizeContext();

    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc<1:0> = bits(2) UNKNOWN;
    else
        // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '0' then
            new_pc<0> = '0';                // T32
        else
            new_pc<1:0> = '00';             // A32

    BranchTo(new_pc, BranchType_UNKNOWN);
```

Library pseudocode for aarch32/functions/system/AArch32.ExecutingATS1xPInstr

```
// AArch32.ExecutingATS1xPInstr()
// ==============================
// Return TRUE if current instruction is AT S1CPR/WP

boolean AArch32.ExecutingATS1xPInstr()
    if !HavePrivATExt() then return FALSE;

    instr = ThisInstr();
    if instr<24+:4> == '1110' && instr<8+:4> == '1110' then
        op1 = instr<21+:3>;
        CRn = instr<16+:4>;
        CRm = instr<0+:4>;
        op2 = instr<5+:3>;
        return (op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000','001'});
    else
        return FALSE;
```

Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

```
// AArch32.ExecutingCP10or11Instr()
// ================================

boolean AArch32.ExecutingCP10or11Instr()
    instr =  ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};

    if instr_set == InstrSet_A32 then
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
    else // InstrSet_T32
        return (instr<31:28> == '111x' && (instr<27:24> == '1110' || instr<27:25> == '110') && instr<11
```

Library pseudocode for aarch32/functions/system/AArch32.ExecutingLSMInstr

```
// AArch32.ExecutingLSMInstr()
// ===========================
// Returns TRUE if processor is executing a Load/Store Multiple instruction

boolean AArch32.ExecutingLSMInstr()
    instr =  ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};

    if instr_set == InstrSet_A32 then
        return (instr<28+:4> != '1111' && instr<25+:3> == '100');
    else // InstrSet_T32
        if ThisInstrLength() == 16 then
            return (instr<12+:4> == '1100');
        else
            return (instr<25+:7> == '1110100' && instr<22> == '0');
```

Library pseudocode for aarch32/functions/system/AArch32.ITAdvance

```
// AArch32.ITAdvance()
// ===================

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead

```
// Read from a 32-bit AArch32 System register and return the register's contents.
bits(32) AArch32.SysRegRead(integer cp_num, bits(32) instr);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead64

```
// Read from a 64-bit AArch32 System register and return the register's contents.
bits(64) AArch32.SysRegRead64(integer cp_num, bits(32) instr);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

```
// AArch32.SysRegReadCanWriteAPSR()
// ================================
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite

```
// Write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, bits(32) val);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite64

```
// Write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, bits(64) val);
```

Library pseudocode for aarch32/functions/system/AArch32.WriteMode

```
// AArch32.WriteMode()
// ===================
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M   = mode;
    PSTATE.EL  = el;
    PSTATE.nRW = '1';
    PSTATE.SP  = (if mode IN {M32_User,M32_System} then '0' else '1');
    return;
```

Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

```
// AArch32.WriteModeByInstr()
// ==========================
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);

    // 'valid' is set to FALSE if' mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```

Library pseudocode for aarch32/functions/system/BadMode

```
// BadMode()
// =========

boolean BadMode(bits(5) mode)
    // Return TRUE if 'mode' encodes a mode that is not valid for this implementation
    case mode of
        when M32_Monitor
            valid = HaveAArch32EL(EL3);
        when M32_Hyp
            valid = HaveAArch32EL(EL2);
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
            valid = HaveAArch32EL(EL1);
        when M32_User
            valid = HaveAArch32EL(EL0);
        otherwise
            valid = FALSE;           // Passed an illegal mode value
    return !valid;
```

Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

    case SYSm of
        when '000xx', '00100'                          // R8_usr to R12_usr
            if mode != M32_FIQ then UNPREDICTABLE;
        when '00101'                                   // SP_usr
            if mode == M32_System then UNPREDICTABLE;
        when '00110'                                   // LR_usr
            if mode IN {M32_Hyp,M32_System} then UNPREDICTABLE;
        when '010xx', '0110x', '01110'                 // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '1000x'                                   // LR_irq, SP_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '1001x'                                   // LR_svc, SP_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '1010x'                                   // LR_abt, SP_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '1011x'                                   // LR_und, SP_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '1110x'                                   // LR_mon, SP_mon
            if !HaveEL(EL3) || !IsSecure() || mode == M32_Monitor then UNPREDICTABLE;
        when '11110'                                   // ELR_hyp, only from Monitor or Hyp mode
            if !HaveEL(EL2) || !(mode IN {M32_Monitor,M32_Hyp}) then UNPREDICTABLE;
        when '11111'                                   // SP_hyp, only from Monitor mode
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// ==================
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;              // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        // Bit <23> is RES0
        if privileged then
            PSTATE.PAN = value<22>;
        // Bits <21:20> are RES0
        PSTATE.GE = value<19:16>;
    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>;                    // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);

    return;
```

Library pseudocode for aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

Library pseudocode for aarch32/functions/system/CurrentCond

```
bits(4) AArch32.CurrentCond();
```

Library pseudocode for aarch32/functions/system/InITBlock

```
// InITBlock()
// ===========

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

Library pseudocode for aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// ==================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    new_spsr = SPSR[];

    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0],J bits

    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>;  // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits

    SPSR[] = new_spsr;                   // UNPREDICTABLE if User or System mode

    return;
```

Library pseudocode for aarch32/functions/system/SPSRaccessValid

```
// SPSRaccessValid()
// =================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110'                                                  // SPSR_fiq
            if mode == M32_FIQ   then UNPREDICTABLE;
        when '10000'                                                  // SPSR_irq
            if mode == M32_IRQ   then UNPREDICTABLE;
        when '10010'                                                  // SPSR_svc
            if mode == M32_Svc   then UNPREDICTABLE;
        when '10100'                                                  // SPSR_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110'                                                  // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100'                                                  // SPSR_mon
            if !HaveEL(EL3) || mode == M32_Monitor || !IsSecure() then UNPREDICTABLE;
        when '11110'                                                  // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

Library pseudocode for aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';

    return;
```

Library pseudocode for aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

Library pseudocode for aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```

Library pseudocode for aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

Library pseudocode for aarch32/translation/attrs/AArch32.DefaultTEXDecode

```
// AArch32.DefaultTEXDecode()
// ==========================

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes memattrs;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits(Unpredictable_RESTEXCB);
        TEX = texcb<4:2>;  C = texcb<1>;  B = texcb<0>;

    case TEX:C:B of
        when '00000'
            // Device-nGnRnE
            memattrs.type = MemType_Device;
            memattrs.device = DeviceType_nGnRnE;
        when '00001', '01000'
            // Device-nGnRE
            memattrs.type = MemType_Device;
            memattrs.device = DeviceType_nGnRE;
        when '00010', '00011', '00100'
            // Write-back or Write-through Read allocate, or Non-cacheable
            memattrs.type = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints(C:B, acctype, FALSE);
            memattrs.outer = ShortConvertAttrsHints(C:B, acctype, FALSE);
            memattrs.shareable = (S == '1');
        when '00110'
            memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
        when '00111'
            // Write-back Read and Write allocate
            memattrs.type = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints('01', acctype, FALSE);
            memattrs.outer = ShortConvertAttrsHints('01', acctype, FALSE);
            memattrs.shareable = (S == '1');
        when '1xxxx'
            // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
            memattrs.type = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints(C:B, acctype, FALSE);
            memattrs.outer = ShortConvertAttrsHints(TEX<1:0>, acctype, FALSE);
            memattrs.shareable = (S == '1');
        otherwise
            // Reserved, handled above
            Unreachable();

    // transient bits are not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;

    // distinction between inner and outer shareable is not supported in this format
    memattrs.outershareable = memattrs.shareable;

    return MemAttrDefaults(memattrs);
```

Library pseudocode for aarch32/translation/attrs/AArch32.InstructionDevice

```
// AArch32.InstructionDevice()
// ===========================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch32.InstructionDevice(AddressDescriptor addrdesc, bits(32) vaddress,
                                            bits(40) ipaddress, integer level,  bits(4) domain,
                                            AccType acctype, boolean iswrite, boolean secondstage,
                                            boolean s2fs1walk)

    c = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, level, acctype, iswrite,
                                                 secondstage, s2fs1walk);
    else
        addrdesc.memattrs.type = MemType_Normal;
        addrdesc.memattrs.inner.attrs = MemAttr_NC;
        addrdesc.memattrs.inner.hints = MemHint_No;
        addrdesc.memattrs.outer = addrdesc.memattrs.inner;
        addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

    return addrdesc;
```

Library pseudocode for aarch32/translation/attrs/AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// ===========================

MemoryAttributes AArch32.RemappedTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes memattrs;

    region = UInt(TEX<0>:C:B);          // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        base = 2 * region;
        attrfield = PRRR<base+1:base>;

        if attrfield == '11' then       // Reserved, maps to allocated value
            (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESPRRR);

        case attrfield of
            when '00'                   // Device-nGnRnE
                memattrs.type = MemType_Device;
                memattrs.device = DeviceType_nGnRnE;
            when '01'                   // Device-nGnRE
                memattrs.type = MemType_Device;
                memattrs.device = DeviceType_nGnRE;
            when '10'
                memattrs.type = MemType_Normal;
                memattrs.inner = ShortConvertAttrsHints(NMRR<base+1:base>, acctype, FALSE);
                memattrs.outer = ShortConvertAttrsHints(NMRR<base+17:base+16>, acctype, FALSE);
                s_bit = if S == '0' then PRRR.NS0 else PRRR.NS1;
                memattrs.shareable = (s_bit == '1');
                memattrs.outershareable = (s_bit == '1' && PRRR<region+24> == '0');
            when '11'
                Unreachable();

    // transient bits are not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;

    return MemAttrDefaults(memattrs);
```

Library pseudocode for aarch32/translation/attrs/AArch32.S1AttrDecode

```
// AArch32.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch32.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;

    if PSTATE.EL == EL2 then
        mair = HMAIR1:HMAIR0;
    else
        mair = MAIR1:MAIR0;
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);

    if attrfield<7:4> == '0000' then            // Device
        memattrs.type = MemType_Device;
        case attrfield<3:0> of
            when '0000'  memattrs.device = DeviceType_nGnRnE;
            when '0100'  memattrs.device = DeviceType_nGnRE;
            when '1000'  memattrs.device = DeviceType_nGRE;
            when '1100'  memattrs.device = DeviceType_GRE;
            otherwise    Unreachable();         // Reserved, handled above

    elsif attrfield<3:0> != '0000'  then        // Normal
        memattrs.type = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';

    else
        Unreachable();                          // Reserved, handled above

    return MemAttrDefaults(memattrs);
```

Library pseudocode for aarch32/translation/attrs/AArch32.TranslateAddressS1Off

```
// AArch32.TranslateAddressS1Off()
// ================================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch32.TranslateAddressS1Off(bits(32) vaddress, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    default_cacheable = (HasS2Translation() && ((if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC) ==

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.type = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;      // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
    elsif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.type = MemType_Device;
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
    else
        // Instruction cacheability controlled by SCTLR/HSCTLR.I
        if PSTATE.EL == EL2 then
            cacheable = HSCTLR.I == '1';
        else
            cacheable = SCTLR.I == '1';
        result.addrdesc.memattrs.type = MemType_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            result.addrdesc.memattrs.inner.hints = MemHint_No;
        result.addrdesc.memattrs.shareable = TRUE;
        result.addrdesc.memattrs.outershareable = TRUE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.physicaladdress = ZeroExtend(vaddress);
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch32.NoFault();
    return result;
```

Library pseudocode for aarch32/translation/checks/AArch32.AccessIsPrivileged

```
// AArch32.AccessIsPrivileged()
// ============================

boolean AArch32.AccessIsPrivileged(AccType acctype)

    if PSTATE.EL == EL0 then
        ispriv = FALSE;
    elsif PSTATE.EL != EL1 then
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType_UNPRIV);

    return ispriv;
```

Library pseudocode for aarch32/translation/checks/AArch32.CheckDomain

```
// AArch32.CheckDomain()
// =====================

(boolean, FaultRecord) AArch32.CheckDomain(bits(4) domain, bits(32) vaddress, integer level,
                                           AccType acctype, boolean iswrite)

    index = 2 * UInt(domain);
    attrfield = DACR<index+1:index>;

    if attrfield == '10' then          // Reserved, maps to an allocated value
        // Reserved value maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESDACR);

    if attrfield == '00' then
        fault = AArch32.DomainFault(domain, level, acctype, iswrite);
    else
        fault = AArch32.NoFault();

    permissioncheck = (attrfield == '01');

    return (permissioncheck, fault);
```

Library pseudocode for aarch32/translation/checks/AArch32.CheckPermission

```
// AArch32.CheckPermission()
// =========================
// Function used for permission checking from AArch32 stage 1 translations

FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level,
                                    bits(4) domain, bit NS, AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());

    if PSTATE.EL != EL2 then
        wxn = SCTLR.WXN == '1';
        if TTBCR.EAE == '1' || SCTLR.AFE == '1' || perms.ap<0> == '1' then
            priv_r = TRUE;
            priv_w = perms.ap<2> == '0';
            user_r = perms.ap<1> == '1';
            user_w = perms.ap<2:1> == '01';
        else
            priv_r = perms.ap<2:1> != '00';
            priv_w = perms.ap<2:1> == '01';
            user_r = perms.ap<1> == '1';
            user_w = FALSE;
        uwxn = SCTLR.UWXN == '1';

        ispriv = AArch32.AccessIsPrivileged(acctype);

        pan = if HavePANExt() then PSTATE.PAN else '0';
        if (pan == '1' && user_r && ispriv &&
            !(acctype IN {AccType_DC,AccType_AT,AccType_IFETCH}) ||
            (acctype == AccType_AT && AArch32.ExecutingATS1xPInstr())) then
            priv_r = FALSE;
            priv_w = FALSE;

        user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
        priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
                   (priv_w && wxn) || (user_w && uwxn));

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2
        wxn = HSCTLR.WXN == '1';
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' then
        secure_instr_fetch = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;
        if secure_instr_fetch == '1' then xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW } then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite && !IsSecure() && PSTATE.EL == EL1 && (acctype != AccType_DC) then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(40) UNKNOWN;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
```

```
    else
        return AArch32.NoFault();
```

Library pseudocode for aarch32/translation/checks/AArch32.CheckS2Permission

```
// AArch32.CheckS2Permission()
// ==========================
// Function used for permission checking from AArch32 stage 2 translations

FaultRecord AArch32.CheckS2Permission(Permissions perms, bits(32) vaddress, bits(40) ipaddress,
                                      integer level, AccType acctype, boolean iswrite,
                                      boolean s2fs1walk)

    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    if HaveExtendedExecuteNeverExt() then
        case perms.xn:perms.xxn of
            when '00'  xn = !r;
            when '01'  xn = !r || PSTATE.EL == EL1;
            when '10'  xn = TRUE;
            when '11'  xn = !r || PSTATE.EL == EL0;
    else
        xn = !r || perms.xn == '1';
    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
        failedread = TRUE;
    elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW }) && !s2fs1walk then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = !iswrite;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch32.NoFault();
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

```
// AArch32.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

    match = FALSE;
    mismatch = FALSE;

    for i = 0 to UInt(DBGDIDR.BRPs)
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_Breakpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

```
// AArch32.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch32.NoFault();

    d_side = (acctype != AccType_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);

    if !d_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(vaddress, size);

    if fault.type == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch32.CheckBreakpoint(vaddress, size);

    if fault.type == Fault_None && !d_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(vaddress, size);

    return fault;
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// ==========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime.
// Vector Catch can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = AArch32.VCRMatch(vaddress);
    if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);

    if match && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_VectorCatch;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = AArch32.AccessIsPrivileged(acctype);

    for i = 0 to UInt(DBGDIDR.WRPs)
        match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elsif match && DBGDSCRext.MDBGen == '1' && AArch32.GenerateDebugExceptions() then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return AArch32.NoFault();
```

Library pseudocode for aarch32/translation/faults/AArch32.AccessFlagFault

```
// AArch32.AccessFlagFault()
// =========================

FaultRecord AArch32.AccessFlagFault(bits(40) ipaddress, bits(4) domain, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_AccessFlag, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.AddressSizeFault

```
// AArch32.AddressSizeFault()
// ==========================

FaultRecord AArch32.AddressSizeFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_AddressSize, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.AlignmentFault

```
// AArch32.AlignmentFault()
// ========================

FaultRecord AArch32.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch32.CreateFaultRecord(Fault_Alignment, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.AsynchExternalAbort

```
// AArch32.AsynchExternalAbort()
// =============================
// Wrapper function for asynchronous external aborts

FaultRecord AArch32.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

    type = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(type, ipaddress,  domain, level, acctype, iswrite, extflag,
                                     debugmoe,  errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.DebugFault

```
// AArch32.DebugFault()
// ====================

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_Debug, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.DomainFault

```
// AArch32.DomainFault()
// =====================

FaultRecord AArch32.DomainFault(bits(4) domain, integer level, AccType acctype, boolean iswrite)

    ipaddress = bits(40) UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_Domain, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.NoFault

```
// AArch32.NoFault()
// =================

FaultRecord AArch32.NoFault()

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_None, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.PermissionFault

```
// AArch32.PermissionFault()
// =========================

FaultRecord AArch32.PermissionFault(bits(40) ipaddress, bits(4) domain, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_Permission, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/faults/AArch32.TranslationFault

```
// AArch32.TranslationFault()
// ==========================

FaultRecord AArch32.TranslationFault(bits(40) ipaddress, bits(4) domain, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch32.CreateFaultRecord(Fault_Translation, ipaddress,  domain, level, acctype, iswrite,
                                     extflag, debugmoe, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch32/translation/translation/AArch32.FirstStageTranslate

```
// AArch32.FirstStageTranslate()
// =============================
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.FirstStageTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                              boolean wasaligned, integer size)

    if PSTATE.EL == EL2 then
        s1_enabled = HSCTLR.M == '1';
    elsif HaveEL(EL2) && !IsSecure() then
        tge = (if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE);
        dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
        s1_enabled = tge == '0' && dc == '0' && SCTLR.M == '1';
    else
        dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
        s1_enabled = dc == '0' && SCTLR.M == '1';

    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                          // First stage enabled
        use_long_descriptor_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
        if use_long_descriptor_format then
            S1 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                                s2fs1walk, size);
            permissioncheck = TRUE;  domaincheck = FALSE;
        else
            S1 = AArch32.TranslationTableWalkSD(vaddress, acctype, iswrite, size);
            permissioncheck = TRUE;  domaincheck = TRUE;
    else
        S1 = AArch32.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;  domaincheck = FALSE;

        if UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() then
            if S1.addrdesc.memattrs.type == MemType_Device && S1.addrdesc.memattrs.device != DeviceType
                nTLSMD = if S1TranslationRegime() == EL2 then HSCTLR.nTLSMD else SCTLR.nTLSMD;
                if nTLSMD == '0' then
                    S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
    // Check for unaligned data accesses to Device memory
    if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
        && S1.addrdesc.memattrs.type == MemType_Device && !IsFault(S1.addrdesc) then
        S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S1.addrdesc) && domaincheck then
        (permissioncheck, abort) = AArch32.CheckDomain(S1.domain, vaddress, S1.level, acctype,
                                                       iswrite);
        S1.addrdesc.fault = abort;

    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch32.CheckPermission(S1.perms, vaddress, S1.level,
                                                    S1.domain, S1.addrdesc.paddress.NS,
                                                    acctype, iswrite);

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
        acctype == AccType_IFETCH) then
        S1.addrdesc = AArch32.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                                S1.domain, acctype, iswrite,
                                                secondstage, s2fs1walk);

    return S1.addrdesc;
```

Library pseudocode for aarch32/translation/translation/AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch32.FullTranslate(bits(32) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch32.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(S1) && HasS2Translation() then
        s2fs1walk = FALSE;
        result = AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                              size);
    else
        result = S1;

    return result;
```

Library pseudocode for aarch32/translation/translation/AArch32.SecondStageTranslate

```
// AArch32.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.SecondStageTranslate(AddressDescriptor S1, bits(32) vaddress,
                                               AccType acctype, boolean iswrite, boolean wasaligned,
                                               boolean s2fs1walk, integer size)
    assert HasS2Translation();
    assert IsZero(S1.paddress.physicaladdress<47:40>);
    hwupdatewalk = FALSE;
    if !ELUsingAArch32(EL2) then
        return AArch64.SecondStageTranslate(S1, ZeroExtend(vaddress, 64), acctype, iswrite,
                                            wasaligned, s2fs1walk, size, hwupdatewalk);

    s2_enabled = HCR.VM == '1' || HCR.DC == '1';
    secondstage = TRUE;

    if s2_enabled then                          // Second stage enabled
        ipaddress = S1.paddress.physicaladdress<39:0>;

        S2 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage,
                                            s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
            && S2.addrdesc.memattrs.type == MemType_Device && !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch32.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                          acctype, iswrite, s2fs1walk);
        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype == AccType_IFETCH) then
            domain = bits(4) UNKNOWN;
            S2.addrdesc = AArch32.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    domain, acctype, iswrite,
                                                    secondstage, s2fs1walk);

        // Check for protected table walk
        if (s2fs1walk && !IsFault(S2.addrdesc) && HCR.PTW == '1' &&
            S2.addrdesc.memattrs.type == MemType_Device) then
            domain = bits(4) UNKNOWN;
            S2.addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, S2.level, acctype,
                                                        iswrite, secondstage, s2fs1walk);

        result = CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

    return result;
```

Library pseudocode for aarch32/translation/translation/AArch32.SecondStageWalk

```
// AArch32.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch32.SecondStageWalk(AddressDescriptor S1, bits(32) vaddress, AccType acctype,
                                          boolean iswrite, integer size)

    assert HasS2Translation();

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                        size);
```

Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)

    if !ELUsingAArch32(S1TranslationRegime()) then
        return AArch64.TranslateAddress(ZeroExtend(vaddress, 64), acctype, iswrite, wasaligned,
                                        size);
    result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
```

Library pseudocode for aarch32/translation/translation/IsEL1TransRegimeRegs

```
// IsEL1TransRegimeRegs()
// ======================

// Returns TRUE if its a register in EL1 translation regime

boolean IsEL1TransRegimeRegs()
    return !HaveEL(EL2) || PSTATE.EL == EL1 || (PSTATE.EL == EL0 && (HCR_EL2.E2H == '0' ||  HCR_EL2.TGE
```

Library pseudocode for aarch32/translation/walk/AArch32.TranslationTableWalkLD

```
// AArch32.TranslationTableWalkLD()
// ================================
// Returns a result of a translation table walk using the Long-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkLD(bits(40) ipaddress, bits(32) vaddress,
                                         AccType acctype, boolean iswrite, boolean secondstage,
                                         boolean s2fs1walk, integer size)
    if !secondstage then
        assert ELUsingAArch32(S1TranslationRegime());
    else
        assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(40) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2
    domain = bits(4) UNKNOWN;

    descaddr.memattrs.type = MemType_Normal;

    // Fixed parameters for the page table walk:
    //  grainsize = Log2(Size of Table)         - Size of Table is 4KB in AArch32
    //  stride = Log2(Address per Level)        - Bits of address consumed at each level
    constant integer grainsize = 12;                    // Log2(4KB page size)
    constant integer stride = grainsize - 3;            // Log2(page size / 8 bytes)

    // Derived parameters for the page table walk:
    //  inputsize = Log2(Size of Input Address) - Input Address size in bits
    //  level = Level to start walk from
    // This means that the number of levels after start level = 3-level

    if !secondstage then
        // First stage translation
        inputaddr = ZeroExtend(vaddress);
        if PSTATE.EL == EL2 then
            inputsize = 32 - UInt(HTCR.T0SZ);
            basefound = inputsize == 32 || IsZero(inputaddr<31:inputsize>);
            disabled = FALSE;
            baseregister = HTTBR;
            descaddr.memattrs = WalkAttrDecode(HTCR.SH0, HTCR.ORGN0, HTCR.IRGN0, secondstage);
            reversedescriptors = HSCTLR.EE == '1';
            lookupsecure = FALSE;
            singlepriv = TRUE;
            hierattrsdisabled = AArch32.HaveHPDExt() && HTCR.HPD == '1';
        else
            basefound = FALSE;
            disabled = FALSE;
            t0size = UInt(TTBCR.T0SZ);
            if t0size == 0 || IsZero(inputaddr<31:(32-t0size)>) then
                inputsize = 32 - t0size;
                basefound = TRUE;
                disabled = TTBCR.EPD0 == '1';
                baseregister = TTBR0;
                descaddr.memattrs = WalkAttrDecode(TTBCR.SH0, TTBCR.ORGN0, TTBCR.IRGN0, secondstage);
                hierattrsdisabled = AArch32.HaveHPDExt() && TTBCR.T2E == '1' &&  TTBCR2.HPD0 == '1';
            t1size = UInt(TTBCR.T1SZ);
            if (t1size == 0 && !basefound) || (t1size > 0 && IsOnes(inputaddr<31:(32-t1size)>)) then
                inputsize = 32 - t1size;
                basefound = TRUE;
                disabled = TTBCR.EPD1 == '1';
                baseregister = TTBR1;
                descaddr.memattrs = WalkAttrDecode(TTBCR.SH1, TTBCR.ORGN1, TTBCR.IRGN1, secondstage);
                hierattrsdisabled = AArch32.HaveHPDExt() && TTBCR.T2E == '1' &&  TTBCR2.HPD1 == '1';
            reversedescriptors = SCTLR.EE == '1';
            lookupsecure = IsSecure();
            singlepriv = FALSE;
```

```
        // The starting level is the number of strides needed to consume the input address
        level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));

    else
        // Second stage translation
        inputaddr = ipaddress;
        inputsize = 32 - SInt(VTCR.T0SZ);
        // VTCR.S must match VTCR.T0SZ[3]
        if VTCR.S != VTCR.T0SZ<3> then
            (-, inputsize) = ConstrainUnpredictableInteger(32-7, 32+8, Unpredictable_RESVTCRS);
        basefound = inputsize == 40 || IsZero(inputaddr<39:inputsize>);
        disabled = FALSE;
        baseregister = VTTBR;
        descaddr.memattrs = WalkAttrDecode(VTCR.IRGN0, VTCR.ORGN0, VTCR.SH0, secondstage);
        reversedescriptors = HSCTLR.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;

        startlevel = UInt(VTCR.SL0);
        level = 2 - startlevel;
        if level <= 0 then basefound = FALSE;

        // Number of entries in the starting level table =
        //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
        startsizecheck = inputsize - ((3 - level)*stride + grainsize); // Log2(Num of entries)

        // Check for starting level table with fewer than 2 entries or longer than 16 pages.
        // Lower bound check is:  startsizecheck < Log2(2 entries)
        //   That is, VTCR.SL0 == '00' and SInt(VTCR.T0SZ) > 1, Size of Input Address < 2^31 bytes
        // Upper bound check is:  startsizecheck > Log2(pagesize/8*16)
        //   That is, VTCR.SL0 == '01' and SInt(VTCR.T0SZ) < -2, Size of Input Address > 2^34 bytes
        if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

    if !basefound || disabled then
        level = 1;            // AArch64 reports this as a level 0 fault
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    if !IsZero(baseregister<47:40>) then
        level = 0;
        result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    // Bottom bound of the Base address is:
    //     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
    // Number of entries in starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
    baseaddress = baseregister<39:baselowerbound>:Zeros(baselowerbound);

    ns_table = if lookupsecure then '0' else '1';
    ap_table = '00';
    xn_table = '0';
    pxn_table = '0';

    addrselecttop = inputsize - 1;

    repeat
        addrselectbottom = (3-level)*stride + grainsize;

        bits(40) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
        descaddr.paddress.physicaladdress = ZeroExtend(baseaddress OR index);
        descaddr.paddress.NS = ns_table;

        // If there are two stages of translation, then the first stage table walk addresses
        // are themselves subject to translation
        if secondstage || !HasS2Translation() then
            descaddr2 = descaddr;
```

```
    else
        descaddr2 = AArch32.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
        // Check for a fault on the stage 2 walk
        if IsFault(descaddr2) then
            result.addrdesc.fault = descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    descaddr2.vaddress = ZeroExtend(vaddress);

    accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
    desc = _Mem[descaddr2, 8, accdesc];

    if reversedescriptors then desc = BigEndianReverse(desc);

    if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
         // Fault (00), Reserved (10), or Block (01) at level 3
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Valid Block, Page, or Table entry
    if desc<1:0> == '01' || level == 3 then                 // Block (01) or Page (11)
        blocktranslate = TRUE;
    else                                                    // Table (11)
        if !IsZero(desc<47:40>) then
            result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        baseaddress = desc<39:grainsize>:Zeros(grainsize);
        if !secondstage then
            // Unpack the upper and lower table attributes
            ns_table    = ns_table    OR desc<63>;
        if !secondstage && !hierattrsdisabled then
            ap_table<1> = ap_table<1> OR desc<62>;       // read-only

            xn_table    = xn_table    OR desc<60>;
            // pxn_table and ap_table[0] apply only in EL1&0 translation regimes
            if !singlepriv then
                pxn_table   = pxn_table   OR desc<59>;
                ap_table<0> = ap_table<0> OR desc<61>;   // privileged

        level = level + 1;
        addrselecttop = addrselectbottom - 1;
        blocktranslate = FALSE;
until blocktranslate;

// Check the output address is inside the supported range
if !IsZero(desc<47:40>) then
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype,
                                                     iswrite, secondstage, s2fs1walk);
    return result;

// Unpack the descriptor into address and upper and lower block attributes
outputaddress = desc<39:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
// Check the access flag
if desc<10> == '0' then
    result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                    iswrite, secondstage, s2fs1walk);
    return result;
xn = desc<54>;
pxn = desc<53>;
contiguousbit = desc<52>;
nG = desc<11>;
sh = desc<9:8>;
ap = desc<7:6>:'1';
memattr = desc<5:2>;                                     // AttrIndx and NS bit in stage 1

result.domain = bits(4) UNKNOWN;                         // Domains not used
```

```
    result.level = level;
    result.blocksize = 2^((3-level)*stride + grainsize);

    // Stage 1 translation regimes also inherit attributes from the tables
    if !secondstage then
        result.perms.xn      = xn OR xn_table;
        result.perms.ap<2>   = ap<2> OR ap_table<1>;          // Force read-only
        // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
        if !singlepriv then
            result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
            result.perms.pxn   = pxn OR pxn_table;
            // Pages from Non-secure tables are marked non-global in Secure EL1&0
            if IsSecure() then
                result.nG = nG OR ns_table;
            else
                result.nG = nG;
        else
            result.perms.ap<1> = '1';
            result.perms.pxn   = '0';
            result.nG          = '0';
        result.perms.ap<0>   = '1';
        result.addrdesc.memattrs = AArch32.S1AttrDecode(sh, memattr<2:0>, acctype);
        result.addrdesc.paddress.NS = memattr<3> OR ns_table;
    else
        result.perms.ap<2:1> = ap<2:1>;
        result.perms.ap<0>   = '1';
        result.perms.xn      = xn;
        if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
        result.perms.pxn     = '0';
        result.nG            = '0';
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
        result.addrdesc.paddress.NS = '1';

    result.addrdesc.paddress.physicaladdress = ZeroExtend(outputaddress);
    result.addrdesc.fault = AArch32.NoFault();
    result.contiguous = contiguousbit == '1';
    if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;
    return result;
```

Library pseudocode for aarch32/translation/walk/AArch32.TranslationTableWalkSD

```
// AArch32.TranslationTableWalkSD()
// ================================
// Returns a result of a translation table walk using the Short-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkSD(bits(32) vaddress, AccType acctype, boolean iswrite,
                                         integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    // This is only called when address translation is enabled
    TLBRecord         result;
    AddressDescriptor l1descaddr;
    AddressDescriptor l2descaddr;
    bits(40)      outputaddress;

    // Variables for Abort functions
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    // Default setting of the domain
    domain = bits(4) UNKNOWN;

    // Determine correct Translation Table Base Register to use.
    bits(64) ttbr;
    n = UInt(TTBCR.N);
    if n == 0 || IsZero(vaddress<31:(32-n)>) then
        ttbr = TTBR0;
        disabled = (TTBCR.PD0 == '1');
    else
        ttbr = TTBR1;
        disabled = (TTBCR.PD1 == '1');
        n = 0;  // TTBR1 translation always works like N=0 TTBR0 translation

    // Check this Translation Table Base Register is not disabled.
    if disabled then
        level = 1;
        result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    // Obtain descriptor from initial lookup.
    l1descaddr.paddress.physicaladdress = ZeroExtend(ttbr<31:14-n>:vaddress<31-n:20>:'00');
    l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
    IRGN = ttbr<0>:ttbr<6>;             // TTBR.IRGN
    RGN = ttbr<4:3>;                    // TTBR.RGN
    SH = ttbr<1>:ttbr<5>;               // TTBR.S:TTBR.NOS
    l1descaddr.memattrs = WalkAttrDecode(SH, RGN, IRGN, secondstage);

    if !HaveEL(EL2) || IsSecure()  then
        // if only 1 stage of translation
        l1descaddr2 = l1descaddr;
    else
        l1descaddr2 = AArch32.SecondStageWalk(l1descaddr, vaddress, acctype, iswrite, 4);
        // Check for a fault on the stage 2 walk
        if IsFault(l1descaddr2) then
            result.addrdesc.fault = l1descaddr2.fault;
            return result;

    // Update virtual address for abort functions
    l1descaddr2.vaddress = ZeroExtend(vaddress);

    accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
    l1desc = _Mem[l1descaddr2, 4,accdesc];

    if SCTLR.EE == '1' then l1desc = BigEndianReverse(l1desc);
```

```
    // Process descriptor from initial lookup.
    case l1desc<1:0> of
        when '00'                                               // Fault, Reserved
            level = 1;
            result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        when '01'                                               // Large page or Small page
            domain = l1desc<8:5>;
            level = 2;
            pxn = l1desc<2>;
            NS = l1desc<3>;

            // Obtain descriptor from level 2 lookup.
            l2descaddr.paddress.physicaladdress = ZeroExtend(l1desc<31:10>:vaddress<19:12>:'00');
            l2descaddr.paddress.NS = if IsSecure() then '0' else '1';
            l2descaddr.memattrs = l1descaddr.memattrs;

            if !HaveEL(EL2) || IsSecure()  then
             // if only 1 stage of translation
                l2descaddr2 = l2descaddr;
            else
                l2descaddr2 = AArch32.SecondStageWalk(l2descaddr, vaddress, acctype, iswrite, 4);
                // Check for a fault on the stage 2 walk
                if IsFault(l2descaddr2) then
                    result.addrdesc.fault = l2descaddr2.fault;
                    return result;

            // Update virtual address for abort functions
            l2descaddr2.vaddress = ZeroExtend(vaddress);

            accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
            l2desc = _Mem[l2descaddr2, 4, accdesc];

            if SCTLR.EE == '1' then l2desc = BigEndianReverse(l2desc);

            // Process descriptor from level 2 lookup.
            if l2desc<1:0> == '00' then
                result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype,
                                                                 iswrite, secondstage, s2fs1walk);
                return result;

            nG = l2desc<11>;
            S = l2desc<10>;
            ap = l2desc<9,5:4>;

            if SCTLR.AFE == '1' && l2desc<4> == '0' then
                // Hardware access to the Access Flag is not supported in ARMv8
                result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                                iswrite, secondstage, s2fs1walk);
                return result;

            if l2desc<1> == '0' then                            // Large page
                xn = l2desc<15>;
                tex = l2desc<14:12>;
                c = l2desc<3>;
                b = l2desc<2>;
                blocksize = 64;
                outputaddress = ZeroExtend(l2desc<31:16>:vaddress<15:0>);
            else                                                // Small page
                tex = l2desc<8:6>;
                c = l2desc<3>;
                b = l2desc<2>;
                xn = l2desc<0>;
                blocksize = 4;
                outputaddress = ZeroExtend(l2desc<31:12>:vaddress<11:0>);

        when '1x'                                               // Section or Supersection
            NS = l1desc<19>;
```

```
            nG = l1desc<17>;
            S = l1desc<16>;
            ap = l1desc<15,11:10>;
            tex = l1desc<14:12>;
            xn = l1desc<4>;
            c = l1desc<3>;
            b = l1desc<2>;
            pxn = l1desc<0>;
            level = 1;

            if SCTLR.AFE == '1' && l1desc<10> == '0' then
                // Hardware management of the Access Flag is not supported in ARMv8
                result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype,
                                                                iswrite, secondstage, s2fs1walk);
                return result;

            if l1desc<18> == '0' then                          // Section
                domain = l1desc<8:5>;
                blocksize = 1024;
                outputaddress = ZeroExtend(l1desc<31:20>:vaddress<19:0>);
            else                                               // Supersection
                domain = '0000';
                blocksize = 16384;
                outputaddress = l1desc<8:5>:l1desc<23:20>:l1desc<31:24>:vaddress<23:0>;

    // Decode the TEX, C, B and S bits to produce the TLBRecord's memory attributes
    if SCTLR.TRE == '0' then
        if RemapRegsHaveResetValues() then
            result.addrdesc.memattrs = AArch32.DefaultTEXDecode(tex, c, b, S, acctype);
        else
            result.addrdesc.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        result.addrdesc.memattrs = AArch32.RemappedTEXDecode(tex, c, b, S, acctype);

    // Set the rest of the TLBRecord, try to add it to the TLB, and return it.
    result.perms.ap = ap;
    result.perms.xn = xn;
    result.perms.pxn = pxn;
    result.nG = nG;
    result.domain = domain;
    result.level = level;
    result.blocksize = blocksize;
    result.addrdesc.paddress.physicaladdress = ZeroExtend(outputaddress);
    result.addrdesc.paddress.NS = if IsSecure() then NS else '1';
    result.addrdesc.fault = AArch32.NoFault();

    return result;
```

Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

```
boolean RemapRegsHaveResetValues();
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.BRPs);

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                     linked, DBGBCR_EL1[n].LBN, isbreakpnt, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAnyAArch32() && size == 4 then                 // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    match = value_match && state_match && enabled;

    return match;
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointValueMatch

```
// 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-existant breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(ID_AA64DFR0_EL1.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(ID_AA64DFR0_EL1.BRPs), Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking.)
    if DBGBCR_EL1[n].E == '0' then return FALSE;

    context_aware = (n >= UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    type = DBGBCR_EL1[n].BT;

    if ((type IN {'011x','11xx'} && !HaveVirtHostExt()) ||              // Context matching
          type == '010x' ||                                             // Reserved
          (type != '0x0x' && !context_aware) ||                         // Context matching
          (type == '1xxx' && !HaveEL(EL2))) then                        // EL2 extension
        (c, type) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = (type == '0x0x');
    match_vmid = (type == '10xx');
    match_cid  = (type == '001x');
    match_cid1 = (type IN { '101x', 'x11x'});
    match_cid2 = (type == '11xx');
    linked     = (type == 'xxx1');

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return FALSE;

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return FALSE;

    // Do the comparison.
    if match_addr then
        byte = UInt(vaddress<1:0>);
        if HaveAnyAArch32() then
            // T32 instructions can be executed at EL0 in an AArch64 translation regime.
            assert byte IN {0,2};                 // "vaddress" is halfword aligned.
            byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0;                     // "vaddress" is word aligned
            byte_select_match = TRUE;             // DBGBCR_EL1[n].BAS<byte> is RES1
        top = AddrTop(vaddress, TRUE, PSTATE.EL);
        BVR_match = vaddress<top:2> == DBGBVR_EL1[n]<top:2> && byte_select_match;
    elsif match_cid then
        if IsInHost() then
            BVR_match = (CONTEXTIDR_EL2 == DBGBVR_EL1[n]<31:0>);
        else
            BVR_match = (PSTATE.EL IN {EL0,EL1} && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
    elsif match_cid1 then
        BVR_match = (PSTATE.EL IN {EL0,EL1} && !IsInHost() && CONTEXTIDR_EL1 == 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 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                      !IsInHost() &&
                      vmid == bvr_vmid);
    elsif match_cid2 then
        BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
                      DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2);

    bvr_match_valid = (match_addr || match_cid || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

    return match;
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.StateMatch

```
// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreakpnt, boolean ispriv)
    // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "linked" is TRUE if this is a linked breakpoint/watchpoint type.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    if ((HMC:SSC:PxC) IN {'011xx','100x0','101x0','11010','11101','1111x'} ||       // Reserved
          (HMC == '0' && PxC == '00' && (!isbreakpnt || !HaveAArch32EL(EL1))) ||    // Usr/Svc/Sys
          (SSC IN {'01','10'} && !HaveEL(EL3)) ||                                   // No EL3
          (HMC:SSC != '000' && HMC:SSC != '111' && !HaveEL(EL3) && !HaveEL(EL2)) || // No EL3/EL2
          (HMC:SSC:PxC == '11100' && !HaveEL(EL2))) then                            // No EL2
        (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
    EL2_match = HaveEL(EL2) && HMC == '1';
    EL1_match = PxC<0> == '1';
    EL0_match = PxC<1> == '1';

    if !ispriv && !isbreakpnt 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 = TRUE;         // Both

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));
        last_ctx_cmp = UInt(ID_AA64DFR0_EL1.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCT
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(64) UNKNOWN;
        linked_to = TRUE;
        linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =================================

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);
```

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = HaveEL(EL2) && !secure && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');
    target = (if route_to_el2 then EL2 else EL1);

    enabled = !HaveEL(EL3) || !secure || MDCR_EL3.SDD == '0';

    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from);

    return enabled;
```

Library pseudocode for aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```
// AArch64.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch64.CheckForPMUOverflow()

    pmuirq = (PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSSET_EL0<31> == '1');
    for n = 0 to UInt(PMCR_EL0.N) - 1
        if HaveEL(EL2) then
            E = (if n < UInt(MDCR_EL2.HPMN) then PMCR_EL0.E else MDCR_EL2.HPME);
        else
            E = PMCR_EL0.E;
        if E == '1' && PMINTENSET_EL1<n> == '1' && PMOVSSET_EL0<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

    return pmuirq;
```

Library pseudocode for aarch64/debug/pmu/AArch64.CountEvents

```
// AArch64.CountEvents()
// =====================
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch64.CountEvents(integer n)
    assert (n == 31 || n < UInt(PMCR_EL0.N));

    // Event counting is disabled in Debug state
    debug = Halted();

    // In Non-secure state, some counters are reserved for EL2
    if HaveEL(EL2) then
        E = (if n < UInt(MDCR_EL2.HPMN) || n == 31 then PMCR_EL0.E else MDCR_EL2.HPME);
    else
        E = PMCR_EL0.E;
    enabled = (E == '1' && PMCNTENSET_EL0<n> == '1');

    if !IsSecure() then
        // Event counting in Non-secure state is allowed unless all of:
        // * EL2 and the HPMD Extension are implemented
        // * Executing at EL2
        // * PMNx is not reserved for EL2
        // * MDCR_EL2.HPMD == 1
        if HaveHPMDExt() && PSTATE.EL == EL2 && (n < UInt(MDCR_EL2.HPMN) || n == 31) then
            prohibited = (MDCR_EL2.HPMD == '1');
        else
            prohibited = FALSE;
    else
        // Event counting in Secure state is prohibited unless any one of:
        // * EL3 is not implemented
        // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
        prohibited = (HaveEL(EL3) && MDCR_EL3.SPME == '0');

    // The IMPLEMENTATION DEFINED authentication interface might override software controls
    if ExternalSecureNoninvasiveDebugEnabled() then prohibited = FALSE;

    // For the cycle counter, PMCR_EL0.DP enables counting when otherwise prohibited
    if prohibited && n == 31 then prohibited = (PMCR_EL0.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M} bits
    filter = (if n == 31 then PMCCFILTR else PMEVTYPER[n]);

    P = filter<31>;
    U = filter<30>;
    NSK = (if HaveEL(EL3) then filter<29> else '0');
    NSU = (if HaveEL(EL3) then filter<28> else '0');
    NSH = (if HaveEL(EL2) then filter<27> else '0');
    M = (if HaveEL(EL3) then filter<26> else '0');

    case PSTATE.EL of
        when EL0  filtered = (if IsSecure() then U == '1' else U != NSU);
        when EL1  filtered = (if IsSecure() then P == '1' else P != NSK);
        when EL2  filtered = (NSH == '0');
        when EL3  filtered = (M != P);

    return !debug && enabled && !prohibited && !filtered;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CheckProfilingBufferAccess

```
// CheckProfilingBufferAccess()
// ============================

SysRegAccess CheckProfilingBufferAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;

    if HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL1 && MDCR_EL2.E2PB<0> != '1' then
        return SysRegAccess_TrapToEL2;

    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;

    return SysRegAccess_OK;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CheckStatisticalProfilingAccess

```
// CheckStatisticalProfilingAccess()
// =================================

SysRegAccess CheckStatisticalProfilingAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;

    if HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL1 && MDCR_EL2.TPMS == '1' then
        return SysRegAccess_TrapToEL2;

    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;

    return SysRegAccess_OK;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// ====================

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if  PSTATE.EL == EL2 then return FALSE;
    if HaveEL(EL2) && !IsSecure() && 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  !HaveEL(EL2) || IsSecure() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// ========================

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    if !secure && HaveEL(EL2) then
        return PMSCR_EL2.PA == '1' && (el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectRecord

```
// CollectRecord()
// ===============

boolean CollectRecord(bits(64) events, integer total_latency, OpType optype)
    assert StatisticalProfilingEnabled();
    if PMSFCR_EL1.FE == '1' then
        e = events<63:48,31:24,15:12,7,5,3,1>;
        m = PMSEVFR_EL1<63:48,31:24,15:12,7,5,3,1>;
        // Check for UNPREDICTABLE case
        if IsZero(PMSEVFR_EL1) && ConstrainUnpredictableBool(Unpredictable_ZEROPMSEVFR) then return FAL
        if !IsZero(NOT(e) AND m) then return FALSE;
    if PMSFCR_EL1.FT == '1' then
        // Check for UNPREDICTABLE case
        if IsZero(PMSFCR_EL1.<B,LD,ST>) && ConstrainUnpredictableBool(Unpredictable_NOOPTYPES) then
            return FALSE;
        case optype of
            when OpType_Branch if PMSFCR_EL1.B == '0' then return FALSE;
            when OpType_Load if PMSFCR_EL1.LD == '0' then return FALSE;
            when OpType_Store if PMSFCR_EL1.ST == '0' then return FALSE;
            when OpType_LoadAtomic if PMSFCR_EL1.<LD,ST> == '00' then return FALSE;
            otherwise return FALSE;
    if PMSFCR_EL1.FL == '1' then
        if IsZero(PMSLATFR_EL1.MINLAT) && ConstrainUnpredictableBool(Unpredictable_ZEROMINLATENCY) then
            return FALSE;
        if total_latency < UInt(PMSLATFR_EL1.MINLAT) then return FALSE;
    return TRUE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// ==================

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (secure, el) = ProfilingBufferOwner();
    if el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp_None;
    if !secure && HaveEL(EL2) then
        pct = PMSCR_EL2.PCT == '1' && (el == EL2 || PMSCR_EL1.PCT == '1');
    else
        pct = PMSCR_EL1.PCT == '1';
    return (if pct then TimeStamp_Physical else TimeStamp_Virtual);
```

Library pseudocode for aarch64/debug/statisticalprofiling/OpType

```
enumeration OpType {
 OpType_Load,                 // Any memory-read operation other than atomics, compare-and-swap, and sw
 OpType_Store,                // Any memory-write operation, including atomics without return
 OpType_LoadAtomic,           // Atomics with return, compare-and-swap and swap
 OpType_Branch,               // Software write to the PC
 OpType_Other                 // Any other class of operation
 };
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// ========================

boolean ProfilingBufferEnabled()
    if !HaveStatisticalProfiling() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    non_secure_bit = if secure then '0' else '1';
    return (!ELUsingAArch32(el) && non_secure_bit == SCR_EL3.NS &&
            PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// ======================

(boolean, bits(2)) ProfilingBufferOwner()
    secure = if HaveEL(EL3) then (MDCR_EL3.NSPB<1> == '0') else IsSecure();
    el = if !secure && HaveEL(EL2) && MDCR_EL2.E2PB == '00' then EL2 else EL1;
    return (secure, el);
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();
```

Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =============================

boolean StatisticalProfilingEnabled()
    if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    in_host = !IsSecure() && HaveEL(EL2) && HCR_EL2.TGE == '1';
    (secure, el) = ProfilingBufferOwner();
    if UInt(el) <  UInt(PSTATE.EL) || secure != IsSecure() || (in_host && el == EL1)  then
        return FALSE;

    case PSTATE.EL of
        when EL3  Unreachable();
        when EL2  spe_bit = PMSCR_EL2.E2SPE;
        when EL1  spe_bit = PMSCR_EL1.E1SPE;
        when EL0  spe_bit = (if in_host then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);

    return spe_bit == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/SysRegAccess

```
enumeration SysRegAccess { SysRegAccess_OK,
                           SysRegAccess_UNDEFINED,
                           SysRegAccess_TrapToEL1,
                           SysRegAccess_TrapToEL2,
                           SysRegAccess_TrapToEL3 };
```

Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```
enumeration TimeStamp { TimeStamp_None,
                        TimeStamp_Virtual,
                        TimeStamp_Physical };
```

Library pseudocode for aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception Level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
    SynchronizeContext();
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    // 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();

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;  PSTATE.nRW = '0';  PSTATE.SP = '1';

    SPSR[] = bits(32) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then                                 // Coming from AArch32
        PSTATE.IT = '00000000';  PSTATE.T = '0';    // PSTATE.J is RES0
    if HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) && SCTLR[].SPAN == '
        PSTATE.PAN = '1';

    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    // SCTLR[].IESB might be ignored in Debug state.
    if HaveRASExt() && SCTLR[].IESB == '1' && ConstrainUnpredictableBool(Unpredictable_IESBinDebug) the
        ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);

    EndOfInstruction();
```

Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// =============================

boolean AArch64.WatchpointByteMatch(integer n, bits(64) vaddress)

    top = AddrTop(vaddress, FALSE, PSTATE.EL);
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                      // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                                         // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;             // Disabled
            when Constraint_NONE      mask = 0;                 // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
        WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

    return WVR_match && byte_select_match;
```

Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
                                boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.WRPs);

    // "ispriv" is FALSE for LDTR/STTR instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, DBGWCR_EL1[n].LBN, isbreakpnt, ispriv);

    ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, 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);
    elsif fault.acctype == AccType_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception type, FaultRecord fault, bits(64) vaddress)

    exception = ExceptionSyndrome(type);

    d_side = type IN {Exception_DataAbort, Exception_Watchpoint};

    exception.syndrome = AArch64.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.ipaddress = fault.ipaddress;
    else
        exception.ipavalid = FALSE;

    return exception;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// ==========================

AArch64.CheckPCAlignment()

    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// ===================

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                    (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault))));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                    (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault))));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// ==========================
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PCAlignment);
    exception.vaddress = ThisInstrAddr();

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SPAlignment);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// ==================================

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
    exception = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0,EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0,EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====================================

AArch64.TakePhysicalSErrorException(boolean impdef_syndrome, bits(24) syndrome)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome<24> = if impdef_syndrome then '1' else '0';
    exception.syndrome<23:0> = syndrome;

    ClearPendingPhysicalSError();

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =================================

AArch64.TakeVirtualFIQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;

    exception = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =================================

AArch64.TakeVirtualIRQException()
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// ====================================

AArch64.TakeVirtualSErrorException(boolean impdef_syndrome, bits(24) syndrome)

    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';  // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome<24> = if impdef_syndrome then '1' else '0';
    exception.syndrome<23:0> = syndrome;

    HCR_EL2.VSE = '0';
    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 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// ============================

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// ===============================

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert HaveEL(EL2) && !IsSecure() && (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 = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// ========================
// Return the Exception Class and Instruction Length fields for reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception type, bits(2) target_el)

    il = if ThisInstrLength() == 32 then '1' else '0';
    from_32 = UsingAArch32();
    assert from_32 || il == '1';            // AArch64 instructions always 32-bit

    case type of
        when Exception_Uncategorized        ec = 0x00; il = '1';
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03;           assert from_32;
        when Exception_CP15RRTTrap          ec = 0x04;           assert from_32;
        when Exception_CP14RTTrap           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_CP14RRTTrap          ec = 0x0C;           assert from_32;
        when Exception_IllegalState         ec = 0x0E; il = '1';
        when Exception_SupervisorCall       ec = 0x11;
        when Exception_HypervisorCall       ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_SystemRegisterTrap   ec = 0x18;           assert !from_32;
        when Exception_InstructionAbort     ec = 0x20; il = '1';
        when Exception_PCAlignment          ec = 0x22; il = '1';
        when Exception_DataAbort            ec = 0x24;
        when Exception_SPAlignment          ec = 0x26; il = '1'; assert !from_32;
        when Exception_FPTrappedException   ec = 0x28;
        when Exception_SError               ec = 0x2F; il = '1';
        when Exception_Breakpoint           ec = 0x30; il = '1';
        when Exception_SoftwareStep         ec = 0x32; il = '1';
        when Exception_Watchpoint           ec = 0x34; il = '1';
        when Exception_SoftwareBreakpoint   ec = 0x38;
        when Exception_VectorCatch          ec = 0x3A; il = '1'; assert from_32;
        otherwise                           Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    return (ec,il);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception type = exception.type;

    (ec,il) = AArch64.ExceptionClass(type, target_el);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = ec<5:0>:il:iss;

    if type IN {Exception_InstructionAbort, Exception_PCAlignment, Exception_DataAbort,
                Exception_Watchpoint} then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if target_el == EL2 then
        if exception.ipavalid then
            HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
        else
            HPFAR_EL2<43:4> = bits(40) UNKNOWN;

    return;
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch64.ResetControlRegisters(boolean cold_reset);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// ===================
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert !HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset the system registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';              // Select stack pointer
    PSTATE.<D,A,I,F>  = '1111';   // All asynchronous exceptions masked
    PSTATE.SS = '0';              // Clear software step bit
    PSTATE.IL = '0';              // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;
    // The reset vector must be correctly aligned
    assert IsZero(rv<63:PAMax()>) && IsZero(rv<1:0>);

    BranchTo(rv, BranchType_UNKNOWN);
```

Library pseudocode for aarch64/exceptions/ieeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// ============================

AArch64.FPTrappedException(boolean is_ase, integer element, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome(Exception_FPTrappedException);
    exception.syndrome<23> = '1';                                  // TFV
    if is_ase then exception.syndrome<10:8> = element<2:0>;    // VECITR
    exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF

    route_to_el2 = HaveEL(EL2) && !IsSecure() && 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 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/takeexception/AArch64.TakeException

```
// 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)
    SynchronizeContext();
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    // 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();

    if UInt(target_el) > UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == EL3 then
            if !IsSecure() && HaveEL(EL2) then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
        elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
            lower_32 = ELUsingAArch32(EL0);
        else
            lower_32 = ELUsingAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

    elsif PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    spsr = GetPSRFromPSTATE();

    if HaveUAOExt() then PSTATE.UAO = '0';
    if !(exception.type IN {Exception_IRQ, Exception_FIQ}) then
        AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;  PSTATE.nRW = '0';  PSTATE.SP = '1';

    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))) && SCTLR[].SPAN == '
        PSTATE.PAN = '1';

    BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION);

    if HaveRASExt() && SCTLR[].IESB == '1' then
        ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    EndOfInstruction();
```

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

```
// AArch64.AArch32SystemAccessTrap()
// =================================
// Trapped AArch32 System register access other than due to CPTR_EL2 or CPACR_EL1.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, bits(32) aarch32_instr)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AArch32SystemAccessTrapSyndrome(aarch32_instr);

    if target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrapSyndrome

```
// AArch64.AArch32SystemAccessTrapSyndrome()
// =========================================
// Return the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr)

    ExceptionRecord exception;
    cpnum = UInt(instr<11:8>);

    bits(20) iss = Zeros();
    if instr<27:24> == '1110' && instr<4> == '1' && instr<31:28> != '1111' then
        // MRC/MCR
        case cpnum of
            when 10    exception = ExceptionSyndrome(Exception_FPIDTrap);
            when 14    exception = ExceptionSyndrome(Exception_CP14RTTrap);
            when 15    exception = ExceptionSyndrome(Exception_CP15RTTrap);
            otherwise  Unreachable();
        iss<19:17> = instr<7:5>;      // opc2
        iss<16:14> = instr<23:21>;    // opc1
        iss<13:10> = instr<19:16>;    // CRn
        if instr<20> == '1' && instr<15:12> == '1111' then // MRC, Rt==15
            iss<9:5> = '11111';
        elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1>   = instr<3:0>;      // CRm
    elsif instr<27:21> == '1100010' && instr<31:28> != '1111' then
        // MRRC/MCRR
        case cpnum of
            when 14    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
            when 15    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
            otherwise  Unreachable();
        iss<19:16> = instr<7:4>;      // opc1
        if instr<19:16> == '1111' then    // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

        if instr<15:12> == '1111' then    // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1>   = instr<3:0>;      // CRm
    elsif instr<27:25> == '110' && instr<31:28> != '1111' then
        // LDC/STC
        assert cpnum == 14;
        exception = ExceptionSyndrome(Exception_CP14DTTrap);
        iss<19:12> = instr<7:0>;      // imm8
        iss<4>     = instr<23>;       // U
        iss<2:1>   = instr<24,21>;    // P,W
        if instr<19:16> == '1111' then    // Rn==15, LDC(Literal addressing)/STC
            iss<9:5> = bits(5) UNKNOWN;
            iss<3>   = '1';
        else
            iss<9:5> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rn
            iss<3>   = '0';
    else
        Unreachable();
    iss<0> = instr<20>;               // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0>  = iss;

    return exception;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// ============================
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && HaveEL(EL2) && !IsSecure() && 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.CheckAArch32SystemAccess

```
// AArch64.CheckAArch32SystemAccess()
// ==================================
// Check AArch32 System register access instruction for enables and disables

AArch64.CheckAArch32SystemAccess(bits(32) instr)
    cp_num = UInt(instr<11:8>);
    assert cp_num IN {14,15};

    // Decode the AArch32 System register access instruction
    if instr<31:28> != '1111' && instr<27:24> == '1110' && instr<4> == '1' then      // MRC/MCR
        cprt = TRUE;  cpdt = FALSE;  nreg = 1;
        opc1 = UInt(instr<23:21>);
        opc2 = UInt(instr<7:5>);
        CRn  = UInt(instr<19:16>);
        CRm  = UInt(instr<3:0>);
    elsif instr<31:28> != '1111' && instr<27:21> == '1100010' then                   // MRRC/MCRR
        cprt = TRUE;  cpdt = FALSE;  nreg = 2;
        opc1 = UInt(instr<7:4>);
        CRm  = UInt(instr<3:0>);
    elsif instr<31:28> != '1111' && instr<27:25> == '110' && instr<22> == '0' then   // LDC/STC
        cprt = FALSE;  cpdt = TRUE;  nreg = 0;
        opc1 = 0;
        CRn  = UInt(instr<15:12>);
    else
        allocated = FALSE;

    //
    // Coarse-grain decode into CP14 or CP15 encoding space. Each of the CPxxxInstrDecode functions
    // returns TRUE if the instruction is allocated at the current Exception level, FALSE otherwise.
    if cp_num == 14 then
        // LDC and STC only supported for c5 in CP14 encoding space
        if cpdt && CRn != 5 then
            allocated = FALSE;
        else
            // Coarse-grained decode of CP14 based on opc1 field
            case opc1 of
                when 0     allocated = CP14DebugInstrDecode(instr);
                when 1     allocated = CP14TraceInstrDecode(instr);
                when 7     allocated = CP14JazelleInstrDecode(instr);    // JIDR only
                otherwise  allocated = FALSE;          // All other values are unallocated

    elsif cp_num == 15 then
        // LDC and STC not supported in CP15 encoding space
        if !cprt then
            allocated = FALSE;
        else
            allocated = CP15InstrDecode(instr);

            // Coarse-grain traps to EL2 have a higher priority than exceptions generated because
            // the access instruction is UNDEFINED
            if AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm) then
                // For a coarse-grain trap, if it is IMPLEMENTATION DEFINED whether an access from
                // Non-secure User mode is UNDEFINED when the trap is disabled, then it is
                // IMPLEMENTATION DEFINED whether the same access is UNDEFINED or generates a trap
                // when the trap is enabled.
                if PSTATE.EL == EL0 && !IsSecure() && !allocated then
                    if boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at NS EL0" then
                        UNDEFINED;
                AArch64.AArch32SystemAccessTrap(EL2, instr);

    else
        allocated = FALSE;

    if !allocated then
        UNDEFINED;

    // If the instruction is not UNDEFINED, it might be disabled or trapped to a higher EL.
    AArch64.CheckAArch32SystemAccessTraps(instr);

    return;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccessEL1Traps

```
// AArch64.CheckAArch32SystemAccessEL1Traps()
// ==========================================
// Check for configurable disables or traps to EL1 or EL2 of an AArch32 System register
// access instruction.

AArch64.CheckAArch32SystemAccessEL1Traps(bits(32) instr)
    assert PSTATE.EL == EL0;

    trap = FALSE;

    // Decode the AArch32 System register access instruction
    (op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

    if cp_num == 14 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 == 0) || // DBGDTRR
            (op == SystemAccessType_DT && CRn == 5 && opc2 == 0)) then                      // DBGDTRR
            trap = !Halted() && MDSCR_EL1.TDCC == '1';

        elsif opc1 == 0 then
            trap = MDSCR_EL1.TDCC == '1';

        elsif opc1 == 1 then
            trap = CPACR[].TTA == '1';

    elsif cp_num == 15 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0) || // PMCR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 1) || // PMCNTE
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 2) || // PMCNTE
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 3) || // PMOVSR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 6) || // PMCEID
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 7) || // PMCEID
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 1) || // PMXEVT
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 14 && opc2 == 3) || // PMOVSS
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 12)) then          // PMEVTY
            trap = PMUSERENR_EL0.EN == '0';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 14 && opc2 == 4 then // PMSW
            trap = PMUSERENR_EL0.EN == '0' && PMUSERENR_EL0.SW == '0';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 0) || // PMC
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 9)) then                          // PMC
            trap = PMUSERENR_EL0.EN == '0' && (write || PMUSERENR_EL0.CR == '0');

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 13 && opc2 == 2) || // PMX
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8 && CRm <= 11)) then  // PM
            trap = PMUSERENR_EL0.EN == '0' && (write || PMUSERENR_EL0.ER == '0');

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 5 then    // P
            trap = PMUSERENR_EL0.EN == '0' && PMUSERENR_EL0.ER == '0';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then /
            trap = CNTKCTL[].EL0PTEN == '0';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 0 && opc2 == 0 then       /
            trap = CNTKCTL[].EL0PTEN == '0' && CNTKCTL[].EL0VCTEN == '0';

        elsif op == SystemAccessType_RRT && opc1 == 1 && CRm == 14 then                               /
            trap = CNTKCTL[].EL0VCTEN == '0';

    if trap then
        AArch64.AArch32SystemAccessTrap(EL1, instr);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccessEL2Traps

```
// AArch64.CheckAArch32SystemAccessEL2Traps()
// ==========================================
// Check for configurable traps to EL2 of an AArch32 System register access instruction.

AArch64.CheckAArch32SystemAccessEL2Traps(bits(32) instr)
    assert HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1};

    trap = FALSE;

    // Decode the AArch32 System register access instruction
    (op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

    if cp_num == 14 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 0) || // DBGDRAR
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 1) ||                         // DBGDRAR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 0) || // DBGDSAR
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 2)) then                      // DBGDSAR
            trap = MDCR_EL2.TDRA == '1' || MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 4) || // DBGO
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 4) ||    // DBGO
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 4) ||    // DBGO
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 4 && opc2 == 4)) then // DBGP
            trap = MDCR_EL2.TDOSA == '1' || MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1';

        elsif opc1 == 0 && (!Halted() || !(op == SystemAccessType_RT && CRn == 0 && CRm == 5 && opc2 ==
            trap = MDCR_EL2.TDA == '1' || MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1';

        elsif opc1 == 1 then
            trap = CPTR_EL2.TTA == '1';

        elsif op == SystemAccessType_RT && opc1 == 7 && CRn == 0 && CRm == 0 && opc2 == 0 then  // JIDR
            trap = HCR_EL2.TID0 == '1';

    elsif cp_num == 15 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 0) ||  // SCTLR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 0) ||  // TTBR0
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 2) ||                          // TTBR0
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 1) ||  // TTBR1
            (op == SystemAccessType_RRT && opc1 == 1 && CRm == 2) ||                          // TTBR1
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 2 && CRm == 0 && opc2 == 2) ||  // TTBCR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 3 && CRm == 0 && opc2 == 0) ||  // DACR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 5 && CRm == 0 && opc2 == 0) ||  // DFSR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 5 && CRm == 0 && opc2 == 1) ||  // IFSR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 6 && CRm == 0 && opc2 == 0) ||  // DFAR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 6 && CRm == 0 && opc2 == 2) ||  // IFAR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 5 && CRm == 1 && opc2 == 0) ||  // ADFSR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 5 && CRm == 1 && opc2 == 1) ||  // AIFSR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 10 && CRm == 2 && opc2 == 0) || // PRRR/M
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 10 && CRm == 2 && opc2 == 1) || // NMRR/M
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 10 && CRm == 3 && opc2 == 0) || // AMAIR0
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 10 && CRm == 3 && opc2 == 1) || // AMAIR1
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 13 && CRm == 0 && opc2 == 1)) then // CON
            trap = if write then HCR_EL2.TVM == '1' else HCR_EL2.TRVM == '1';
        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 8 then                         // TLBI
            trap = write && HCR_EL2.TTLB == '1';
        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 6 && opc2 == 2) ||  // DCI
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 10 && opc2 == 2) ||    // DCC
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 14 && opc2 == 2)) then // DCC
            trap = write && HCR_EL2.TSW == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 6 && opc2 == 1) ||  // DCI
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 10 && opc2 == 1) ||    // DCC
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 14 && opc2 == 1)) then // DCC
            trap = write && HCR_EL2.TPC == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 1) ||  // ICI
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 5 && opc2 == 0) ||     // ICI
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 1 && opc2 == 0) ||     // ICI
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 11 && opc2 == 1)) then // DCC
```

```
            trap = write && HCR_EL2.TPU == '1';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 1 then   // ACT
            trap = HCR_EL2.TACR == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 2) ||  // TCM
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 3) ||     // TLB
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 6) ||     // REV
            (op == SystemAccessType_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 7)) then  // AID
            trap = HCR_EL2.TID1 == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 0 && opc2 == 1) ||  // CTR
            (op == SystemAccessType_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 0) ||     // CCS
            (op == SystemAccessType_RT && opc1 == 1 && CRn == 0 && CRm == 0 && opc2 == 1) ||     // CLI
            (op == SystemAccessType_RT && opc1 == 2 && CRn == 0 && CRm == 0 && opc2 == 0)) then  // CSS
            trap = HCR_EL2.TID2 == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 1) ||               // ID_
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 2 && opc2 <= 5) ||     // ID_
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm >= 3 && opc2 <= 1) ||     // Res
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 3 && opc2 == 2) ||     // Res
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 0 && CRm == 5 && opc2 IN {4,5})) then  //
            trap = HCR_EL2.TID3 == '1';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 2 then   // CPA
            trap = CPTR_EL2.TCPAC == '1';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm == 12 && opc2 == 0 then  // PMC
            trap = MDCR_EL2.TPMCR == '1' || MDCR_EL2.TPM == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8) ||      // PMEVCNTR<n>
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm IN {12,13,14}) || // PM*
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 9)) then                  // PMCCNTR (MR
            trap = MDCR_EL2.TPM == '1';

        elsif op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm == 2 && opc2 IN {0,1,2} then
            if !HaveVirtHostExt() || HCR_EL2.E2H == '0' then
                trap = CNTHCTL_EL2.EL1PCEN == '0';
            else
                trap = CNTHCTL_EL2.EL1PTEN == '0';
        elsif op == SystemAccessType_RRT && opc1 == 0 && CRm == 14 then
            trap = CNTHCTL_EL2.EL1PCTEN == '0';

    if trap then
        AArch64.AArch32SystemAccessTrap(EL2, instr);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccessEL3Traps

```
// AArch64.CheckAArch32SystemAccessEL3Traps()
// ==========================================
// Check for configurable traps to EL3 of an AArch32 System register access instruction.

AArch64.CheckAArch32SystemAccessEL3Traps(bits(32) instr)
    assert HaveEL(EL3) && PSTATE.EL != EL3;

    // Decode the AArch32 System register access instruction
    (op, cp_num, opc1, CRn, CRm, opc2, write) = AArch32.DecodeSysRegAccess(instr);

    trap = FALSE;

    if cp_num == 14 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 4 && !write) ||
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 4 && write) ||
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 4) ||
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 4 && opc2 == 4)) then
            trap = MDCR_EL3.TDOSA == '1';

        elsif opc1 == 0 && (!Halted() || !(op == SystemAccessType_RT && CRn == 0 && CRm == 5 && opc2 ==
            trap = MDCR_EL3.TDA == '1';

        elsif opc1 == 1 then
            trap = CPTR_EL3.TTA == '1';

    elsif cp_num == 15 then
        if ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 0) ||     // SCR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 1 && opc2 == 2) ||     // NSAC
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 12 && CRm == 0 && opc2 == 1) ||    // MVBA
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 3 && opc2 == 1) ||     // SDCR
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 7 && CRm == 8 && opc2 >= 4)) then // ATS1
            trap = PSTATE.EL == EL1 && IsSecure();

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 1 && CRm == 0 && opc2 == 2) || // CPAC
            (op == SystemAccessType_RT && opc1 == 4 && CRn == 1 && CRm == 1 && opc2 == 2)) then // HCPT
            trap = CPTR_EL3.TCPAC == '1';

        elsif ((op == SystemAccessType_RT && opc1 == 0 && CRn == 14 && CRm >= 8) ||              // PMEV
            (op == SystemAccessType_RT && opc1 == 0 && CRn == 9 && CRm IN {12,13,14}) ||         // PM*
            (op == SystemAccessType_RRT && opc1 == 0 && CRm == 9)) then                          // PMCC
            trap = MDCR_EL3.TPM == '1';

    if trap then
        AArch64.AArch32SystemAccessTrap(EL3, instr);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckAArch32SystemAccessTraps

```
// AArch64.CheckAArch32SystemAccessTraps()
// =======================================
// Check for configurable disables or traps to a higher EL of an AArch32 System register access.

AArch64.CheckAArch32SystemAccessTraps(bits(32) instr)

    if PSTATE.EL == EL0 then
        AArch64.CheckAArch32SystemAccessEL1Traps(instr);
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        AArch64.CheckAArch32SystemAccessEL2Traps(instr);
    AArch64.CheckAArch32SystemAccessEL3Traps(instr);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32 CP15 traps in HSTR_EL2 and HCR_EL2.

boolean AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR_EL2<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR_EL2.TIDCP
        if (HCR_EL2.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
             (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// ===============================
// Check against CPACR[]

AArch64.CheckFPAdvSIMDEnabled()
    if PSTATE.EL IN {EL0, EL1} then
        // Check if access disabled in CPACR_EL1
        case CPACR[].FPEN of
            when 'x0'  disabled = TRUE;
            when '01'  disabled = PSTATE.EL == EL0;
            when '11'  disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap();               // Also check against CPTR_EL2 and CPTR_EL3
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()

    if HaveEL(EL2) && !IsSecure() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.FPEN of
                when 'x0'  disabled = !(PSTATE.EL == EL1 && HCR_EL2.TGE == '1');
                when '01'  disabled = (PSTATE.EL == EL0 && HCR_EL2.TGE == '1');
                when '11'  disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

    return;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

```
// AArch64.CheckForERetTrap()
// ==========================
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    // Non-secure EL1 execution of ERET, ERETAA, ERETAB when HCR_EL2.NV bit is set, is trapped to EL2
    route_to_el2 = (HaveNVExt() && HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL1 && HCR_EL2.NV == '1')

    if route_to_el2 then
        ExceptionRecord exception;
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_ERetTrap);
        exception.syndrome<24:2> = ZeroExtend('0', 23);    // syndrome<24:2> is RES0
        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 UnallocatedEncoding();
    if !HaveEL(EL3) then
        if HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL1 then
            if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UnallocatedEncoding();
        else
            UnallocatedEncoding();
    else
        route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL1 && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFxTrap

```
// AArch64.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    case target_el of
        when EL1 trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2 trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3 trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
    if trap then
        AArch64.WFxTrap(target_el, is_wfe);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()

    if PSTATE.IL == '1' then
        route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;

        exception = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemRegisterTrap

```
// AArch64.SystemRegisterTrap()
// ============================
// Trapped system register access other than due to CPTR_EL2 and CPACR_EL1

AArch64.SystemRegisterTrap(bits(2) target_el, bits(2) op0, bits(3) op2, bits(3) op1, bits(4) crn,
                           bits(5) rt, bits(4) crm, bit dir)
    assert UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
    exception.syndrome<21:20> = op0;
    exception.syndrome<19:17> = op2;
    exception.syndrome<16:14> = op1;
    exception.syndrome<13:10> = crn;
    exception.syndrome<9:5>   = rt;
    exception.syndrome<4:1>   = crm;
    exception.syndrome<0>     = dir;

    if target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.UndefinedFault

```
// AArch64.UndefinedFault()
// ========================

AArch64.UndefinedFault()

    route_to_el2 = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL0 && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =================

AArch64.WFxTrap(bits(2) target_el, boolean is_wfe)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<0> = if is_wfe then '1' else '0';

    if target_el == EL1 && HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =========================
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

Library pseudocode for aarch64/functions/aborts/AArch64.CreateFaultRecord

```
// AArch64.CreateFaultRecord()
// ===========================

FaultRecord AArch64.CreateFaultRecord(Fault type, bits(52) ipaddress,
                                      integer level, AccType acctype, boolean write, bit extflag,
                                      bits(2) errortype, boolean secondstage, boolean s2fs1walk)

    FaultRecord fault;
    fault.type = type;
    fault.domain = bits(4) UNKNOWN;         // Not used from AArch64
    fault.debugmoe = bits(4) UNKNOWN;       // Not used from AArch64
    fault.errortype = errortype;
    fault.ipaddress = ipaddress;
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fs1walk = s2fs1walk;

    return fault;
```

Library pseudocode for aarch64/functions/aborts/AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception Level using AArch64.

bits(25) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.type != Fault_None;

    bits(25) iss = Zeros();
    if HaveRASExt() && IsExternalSyncAbort(fault) then iss<12:11> = fault.errortype; // SET
    if d_side then
        if IsSecondStage(fault) && !fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.type, fault.level);

    return iss;
```

Library pseudocode for aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// ===============================

// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = (address == Align(address, size));

    if !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

Library pseudocode for aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// ==============================

// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = (address != Align(address, size));
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.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;
    op1 = FPNeg(op1);
    (type1,sign1,value1) = FPUnpack(op1, FPCR);
    (type2,sign2,value2) = FPUnpack(op2, FPCR);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```

Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// ==================

bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1,sign1,value1) = FPUnpack(op1, FPCR);
    (type2,sign2,value2) = FPUnpack(op2, FPCR);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0');
        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 FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```

Library pseudocode for aarch64/functions/memory/AArch64.CheckAlignment

```
// AArch64.CheckAlignment()
// ========================

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype,
                               boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED };
    vector  = acctype == AccType_VEC;
    check = (atomic || ordered || SCTLR[].A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

Library pseudocode for aarch64/functions/memory/AArch64.MemSingle

```
// AArch64.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    value = _Mem[memaddrdesc, size, accdesc];
    return value;

// AArch64.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8) v
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.mematrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    _Mem[memaddrdesc, size, accdesc] = value;
    return;
```

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;
```

Library pseudocode for aarch64/functions/memory/Mem

```
// Mem[] - non-assignment (read) form
// ==================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    integer i;
    boolean iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then
        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        value<63:0>   = AArch64.MemSingle[address,   8, acctype, aligned];
        value<127:64> = AArch64.MemSingle[address+8, 8, acctype, aligned];
    else
        value = AArch64.MemSingle[address, size, acctype, aligned];

    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem[] - assignment (write) form
// ===============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
    integer i;
    boolean iswrite = TRUE;

    if BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
```

```
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        AArch64.MemSingle[address,   8, acctype, aligned] = value<63:0>;
        AArch64.MemSingle[address+8, 8, acctype, aligned] = value<127:64>;
    else
        AArch64.MemSingle[address, size, acctype, aligned] = value;
    return;
```

Library pseudocode for aarch64/functions/pac/addpac/AddPAC

```
// AddPAC()
// ========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean tbi = CalculateTBI(ptr, data);
    integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        if IsEL1TransRegimeRegs() then
            if data then
                selbit = if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then ptr<55> else ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
        else
            // EL2 translation regime registers
            if data then
                selbit = if ((HaveEL(EL2) && TCR_EL2.TBI1 == '1') ||
                             (HaveEL(EL2) && TCR_EL2.TBI0 == '1')) then ptr<55> else ptr<63>;
            else
                selbit = if ((HaveEL(EL2) && TCR_EL2.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                             (HaveEL(EL2) && TCR_EL2.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then ptr<55>
    else selbit = if tbi then ptr<55> else ptr<63>;

    integer bottom_PAC_bit = CalculateBottomPACBit(ptr, selbit);

    // The pointer authentication code field takes all the available bits in between
    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

    // Check if the ptr has good extension bits and corrupt the pointer authentication code if not;
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        PAC<top_bit-1> = NOT(PAC<top_bit-1>);

    // Preserve the determination between upper and lower address at bit<55> and insert PAC
    if tbi then
        result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    return result;
```

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 = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_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 = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_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
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = SCR_EL3.API == '0';
        when EL1
            TrapEL2 = HaveEL(EL2) && !IsSecure() && 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 = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_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 = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APIBKey_EL1, FALSE);
```

Library pseudocode for aarch64/functions/pac/auth/Auth

```
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit keynumber)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = CalculateTBI(ptr, data);
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr, ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    if tbi then
        original_ptr = ptr<63:56>:extfield<56-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<64-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
            result = original_ptr;
        else
            error_code = keynumber:NOT(keynumber);
            result = original_ptr<63:55>:error_code:original_ptr<52:0>;
    else
        if ((PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit>) &&
            (PAC<63:56> == ptr<63:56>)) then
            result = original_ptr;
        else
            error_code = keynumber:NOT(keynumber);
            result = original_ptr<63>:error_code:original_ptr<60:0>;
    return result;
```

Library pseudocode for aarch64/functions/pac/authda/AuthDA

```
// AuthDA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APDAKey_EL1, TRUE, '0');
```

Library pseudocode for aarch64/functions/pac/authdb/AuthDB

```
// AuthDB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APDBKey_EL1, TRUE, '1');
```

Library pseudocode for aarch64/functions/pac/authia/AuthIA

```
// AuthIA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APIAKey_EL1, FALSE, '0');
```

Library pseudocode for aarch64/functions/pac/authib/AuthIB

```
// AuthIB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APIBKey_EL1, FALSE, '1');
```

Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =======================

integer CalculateBottomPACBit(bits(64) ptr, bit top_bit)
    integer tsz_field;

    if PtrHasUpperAndLowerAddRanges() then
        if IsEL1TransRegimeRegs() then
            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 == '11';
        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 == '11';
    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 == '11' else TCR_EL3.TG0 == '11';

    max_limit_tsz_field = 39;
    if tsz_field > max_limit_tsz_field then
        // TCR_ELx.TySZ is out of range
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
    tszmin = if using64k && VAMax() == 52 then 12 else 16;
    if tsz_field < tszmin then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = tszmin;
    return (64-tsz_field);
```

Library pseudocode for aarch64/functions/pac/calculatetbi/CalculateTBI

```
// CalculateTBI()
// ==============

boolean CalculateTBI(bits(64) ptr, boolean data)
    boolean tbi = FALSE;

    if PtrHasUpperAndLowerAddRanges() then
        if IsEL1TransRegimeRegs() then
            if data then
                tbi = if ptr<55> == '1' then TCR_EL1.TBI1 == '1' else TCR_EL1.TBI0 == '1';
            else
                if ptr<55> == '1' then
                    tbi = TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0';
                else
                    tbi = TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0';
        else
            // EL2 translation regime registers
            if data then
                tbi = if ptr<55> == '1' then TCR_EL2.TBI1 == '1' else TCR_EL2.TBI0 == '1';
            else
                if ptr<55> == '1' then
                    tbi = TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0';
                else
                    tbi = TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0';
    elsif PSTATE.EL == EL2 then
        tbi = if data then TCR_EL2.TBI=='1' else TCR_EL2.TBI=='1' && TCR_EL2.TBID=='0';
    elsif PSTATE.EL == EL3 then
        tbi = if data then TCR_EL3.TBI=='1' else TCR_EL3.TBI=='1' && TCR_EL3.TBID=='0';

    return tbi;
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```
array bits(64) RC[0..4];

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
    bits(64)  workingval;
    bits(64)  runningmod;
    bits(64)  roundkey;
    bits(64)  modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;
    RC[3] = 0x082EFA98EC4E6C89<63:0>;
    RC[4] = 0x452821E638D01377<63:0>;

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;
    for i = 0 to 4
        roundkey = key1 EOR runningmod;
        workingval  = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
        workingval = PACSub(workingval);
        runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = PACSub(workingval);
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 EOR workingval;
    workingval = PACCellInvShuffle(workingval);
    workingval = PACInvSub(workingval);
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    workingval = workingval EOR key0;
    workingval = workingval EOR runningmod;
    for i = 0 to 4
        workingval = PACInvSub(workingval);
        if i < 4 then
            workingval = PACMult(workingval);
            workingval = PACCellInvShuffle(workingval);
        runningmod = TweakInvShuffle(runningmod<63:0>);
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR RC[4-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```
// PACCellInvShuffle()
// ===================

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

```
// PACCellShuffle()
// ================

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// ===========

bits(64) PACInvSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher

    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '0101';
            when '0001'  Toutput<4*i+3:4*i> = '1110';
            when '0010'  Toutput<4*i+3:4*i> = '1101';
            when '0011'  Toutput<4*i+3:4*i> = '1000';
            when '0100'  Toutput<4*i+3:4*i> = '1010';
            when '0101'  Toutput<4*i+3:4*i> = '1011';
            when '0110'  Toutput<4*i+3:4*i> = '0001';
            when '0111'  Toutput<4*i+3:4*i> = '1001';
            when '1000'  Toutput<4*i+3:4*i> = '0010';
            when '1001'  Toutput<4*i+3:4*i> = '0110';
            when '1010'  Toutput<4*i+3:4*i> = '1111';
            when '1011'  Toutput<4*i+3:4*i> = '0000';
            when '1100'  Toutput<4*i+3:4*i> = '0100';
            when '1101'  Toutput<4*i+3:4*i> = '1100';
            when '1110'  Toutput<4*i+3:4*i> = '0111';
            when '1111'  Toutput<4*i+3:4*i> = '0011';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)
    bits(4)  t0;
    bits(4)  t1;
    bits(4)  t2;
    bits(4)  t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '1011';
            when '0001'  Toutput<4*i+3:4*i> = '0110';
            when '0010'  Toutput<4*i+3:4*i> = '1000';
            when '0011'  Toutput<4*i+3:4*i> = '1111';
            when '0100'  Toutput<4*i+3:4*i> = '1100';
            when '0101'  Toutput<4*i+3:4*i> = '0000';
            when '0110'  Toutput<4*i+3:4*i> = '1001';
            when '0111'  Toutput<4*i+3:4*i> = '1110';
            when '1000'  Toutput<4*i+3:4*i> = '0011';
            when '1001'  Toutput<4*i+3:4*i> = '0111';
            when '1010'  Toutput<4*i+3:4*i> = '0100';
            when '1011'  Toutput<4*i+3:4*i> = '0101';
            when '1100'  Toutput<4*i+3:4*i> = '1101';
            when '1101'  Toutput<4*i+3:4*i> = '0010';
            when '1110'  Toutput<4*i+3:4*i> = '0001';
            when '1111'  Toutput<4*i+3:4*i> = '1010';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =========

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =================

bits(4) TweakCellInvRot(bits(4)incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =================

bits(64) TweakInvShuffle(bits(64)indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// ==============

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
```

Library pseudocode for aarch64/functions/pac/pac/HavePACExt

```
// HavePACExt()
// ============

boolean HavePACExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// ==============================

// Returns TRUE if the pointer has upper and lower address ranges

boolean PtrHasUpperAndLowerAddRanges()
    return PSTATE.EL == EL1 || PSTATE.EL == EL0 || (PSTATE.EL == EL2 && HCR_EL2.E2H == '1');
```

Library pseudocode for aarch64/functions/pac/strip/Strip

```
// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = CalculateTBI(A, data);
    integer bottom_PAC_bit = CalculateBottomPACBit(A, A<55>);
    extfield = Replicate(A<55>, 64);

    if tbi then
        original_ptr = A<63:56>:extfield< 56-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield< 64-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return original_ptr;
```

Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el !=  EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

```
// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()

    route_to_el3 = (HaveEL(EL3) && SCR_EL3.EA == '1');
    route_to_el2 = (HaveEL(EL2) && !IsSecure() &&
                    (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');
    intdis      = (Halted() || ExternalDebugInterruptsDisabled(target));
    masked      = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending
    if SErrorPending() && masked then       // ISR_EL1.A is set to 1
        // 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 HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = (HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1');
    vSEI_pending = (vSEI_enabled && HCR_EL2.VSE == '1');
    vintdis      = (Halted() || ExternalDebugInterruptsDisabled(EL1));
    vmasked      = (vintdis || PSTATE.A == '1');

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if ELUsingAArch32(EL1) then
            VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        else
            VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>);
        HCR_EL2.VSE = '0';                        // Clear pending virtual SError

    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} && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[n]<63:32> = Zeros();

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// ===============================

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i] = bits(64) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// ==============================

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// ===============================

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;
    ELR_EL1  = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(32) UNKNOWN;
        ELR_EL2  = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(32) UNKNOWN;
        ELR_EL3  = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq = bits(32) UNKNOWN;
        SPSR_irq = bits(32) UNKNOWN;
        SPSR_abt = bits(32) UNKNOWN;
        SPSR_und = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSystemRegisters

```
AArch64.ResetSystemRegisters(boolean cold_reset);
```

Library pseudocode for aarch64/functions/registers/PC

```
// PC - non-assignment form
// ========================
// Read program counter.

bits(64) PC[]
    return _PC;
```

Library pseudocode for aarch64/functions/registers/SP

```
// SP[] - assignment form
// ======================
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
    assert width IN {32,64};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value);
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = ZeroExtend(value);
            when EL1  SP_EL1 = ZeroExtend(value);
            when EL2  SP_EL2 = ZeroExtend(value);
            when EL3  SP_EL3 = ZeroExtend(value);
    return;

// SP[] - non-assignment form
// ==========================
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
    assert width IN {8,16,32,64};
    if PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0<width-1:0>;
            when EL1  return SP_EL1<width-1:0>;
            when EL2  return SP_EL2<width-1:0>;
            when EL3  return SP_EL3<width-1:0>;
```

Library pseudocode for aarch64/functions/registers/V

```
// V[] - assignment form
// =====================
// 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};
    _V[n] = ZeroExtend(value);
    return;

// 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 _V[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 IN {8,16,32,64};
        return _V[n]<width-1:0>;
    else
        assert width == 64;
        return _V[n]<(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 IN {8,16,32,64};
        _V[n] = ZeroExtend(value);
    else
        assert width == 64;
        _V[n]<(width * 2)-1:width> = value<width-1:0>;
```

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/sysregisters/CNTKCTL

```
// CNTKCTL[] - non-assignment form
// ===============================

CNTKCTLType CNTKCTL[]
    if IsInHost() then
        return CNTHCTL_EL2;
    return CNTKCTL_EL1;
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

Library pseudocode for aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// =============================

CPACRType CPACR[]
    if IsInHost() then
        return CPTR_EL2;
    return CPACR_EL1;
```

Library pseudocode for aarch64/functions/sysregisters/CPACRType

```
type CPACRType;
```

Library pseudocode for aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// ===========================

bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        when EL3  r = ELR_EL3;
        otherwise Unreachable();
    return r;

// ELR[] - non-assignment form
// ===========================

bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// =======================

ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1  ELR_EL1 = r;
        when EL2  ELR_EL2 = r;
        when EL3  ELR_EL3 = r;
        otherwise Unreachable();
    return;

// ELR[] - assignment form
// =======================

ELR[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR[PSTATE.EL] = value;
    return;
```

Library pseudocode for aarch64/functions/sysregisters/ESR

```
// ESR[] - non-assignment form
// ===========================

ESRType ESR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        when EL3  r = ESR_EL3;
        otherwise Unreachable();
    return r;

// ESR[] - non-assignment form
// ===========================

ESRType ESR[]
    return ESR[S1TranslationRegime()];

// ESR[] - assignment form
// =======================

ESR[bits(2) regime] = ESRType value
    bits(32) r = value;
    case regime of
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        when EL3  ESR_EL3 = r;
        otherwise Unreachable();
    return;

// ESR[] - assignment form
// =======================

ESR[] = ESRType value
    ESR[S1TranslationRegime()] = value;
```

Library pseudocode for aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

Library pseudocode for aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// ===========================

bits(64) FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = FAR_EL1;
        when EL2  r = FAR_EL2;
        when EL3  r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR[] - non-assignment form
// ===========================

bits(64) FAR[]
    return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// =======================

FAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1  FAR_EL1 = r;
        when EL2  FAR_EL2 = r;
        when EL3  FAR_EL3 = r;
        otherwise Unreachable();
    return;

// FAR[] - assignment form
// =======================

FAR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;
```

Library pseudocode for aarch64/functions/sysregisters/MAIR

```
// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = MAIR_EL1;
        when EL2  r = MAIR_EL2;
        when EL3  r = MAIR_EL3;
        otherwise Unreachable();
    return r;

// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[]
    return MAIR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

Library pseudocode for aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// ============================

SCTLRType SCTLR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        when EL3  r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// ============================

SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

Library pseudocode for aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/system/AArch64.CheckAdvSIMDFPSystemRegisterTraps

```
// Checks if an AArch64 MSR, MRS or SYS instruction on a SIMD or floating-point
// register is trapped under the current configuration. Returns a boolean which
// is TRUE if trapping occurs, plus a binary value that specifies the Exception
// level trapped to.
(boolean, bits(2)) AArch64.CheckAdvSIMDFPSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bit
```

Library pseudocode for aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// ===========================

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bits(5) rt,
    // Checks if an AArch64 MSR, MRS or SYS instruction is UNALLOCATED or trapped at the current
    // exception level, security state and configuration, based on the opcode's encoding.
    boolean unallocated = FALSE;
    boolean need_secure = FALSE;
    bits(2) min_EL;

    // Check for traps by HCR_EL2.TIDCP
    if HaveEL(EL2) && !IsSecure() && HCR_EL2.TIDCP == 1 && op0 == 'x1' && crn == '1x11' then
        // At Non-secure EL0, it is IMPLEMENTATION_DEFINED whether attempts to execute system
        // register access instructions with reserved encodings are trapped to EL2 or UNDEFINED
        rcs_el0_trap = boolean IMPLEMENTATION_DEFINED "Reserved Control Space EL0 Trapped";
        if PSTATE.EL == EL0 && rcs_el0_trap then
            AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);
        elsif PSTATE.EL == EL1 then
            AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);

    // Check for unallocated encodings
    case op1 of
        when '00x', '010'
            min_EL = EL1;
        when '011'
            min_EL = EL0;
        when '100'
            min_EL = EL2;
        when '101'
            if !HaveVirtHostExt() then UnallocatedEncoding();
            min_EL = EL2;
        when '110'
            min_EL = EL3;
        when '111'
            min_EL = EL1;
            need_secure = TRUE;

    if UInt(PSTATE.EL) < UInt(min_EL) then
        // Check for traps on read/write access to registers named _EL2, _EL02, _EL12 from non-secure E
        if ( PSTATE.EL == EL1 && min_EL == EL2 && HaveNVExt() && !IsSecure() && HaveEL(EL2) && HCR_EL2.
            AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);
        else
            UnallocatedEncoding();
    elsif need_secure && !IsSecure() then
        UnallocatedEncoding();
    elsif AArch64.CheckUnallocatedSystemAccess(op0, op1, crn, crm, op2, read) then
        UnallocatedEncoding();

    // Check for traps on accesses to SIMD or floating-point registers
    (take_trap, target_el) = AArch64.CheckAdvSIMDFPSystemRegisterTraps(op0, op1, crn, crm, op2, read);
    if take_trap then
        AArch64.AdvSIMDFPAccessTrap(target_el);

    // Check for traps on access to all other system registers
    (take_trap, target_el) = AArch64.CheckSystemRegisterTraps(op0, op1, crn, crm, op2, read);

    if take_trap then
        AArch64.SystemRegisterTrap(target_el, op0, op2, op1, crn, rt, crm, read);
```

Library pseudocode for aarch64/functions/system/AArch64.CheckSystemRegisterTraps

```
// Checks if an AArch64 MSR, MRS or SYS instruction on a system register is trapped
// under the current configuration. Returns a boolean which is TRUE if trapping
// occurs, plus a binary value that specifies the Exception level trapped to.
(boolean, bits(2)) AArch64.CheckSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
```

Library pseudocode for aarch64/functions/system/AArch64.CheckUnallocatedSystemAccess

```
// Checks if an AArch64 MSR, MRS or SYS instruction is unallocated under the current
// configuration.
boolean AArch64.CheckUnallocatedSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingATS1xPInstr

```
// AArch64.ExecutingATS1xPInstr()
// ==============================
// Return TRUE if current instruction is AT S1E1R/WP

boolean AArch64.ExecutingATS1xPInstr()
    if !HavePrivATExt() then return FALSE;

    instr = ThisInstr();
    if instr<22+:10> == '1101010100' then
        op1  = instr<16+:3>;
        CRn  = instr<12+:4>;
        CRm  = instr<8+:4>;
        op2  = instr<5+:3>;
        return (op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000','001'});
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/AArch64.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) System_Get(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/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)

    SynchronizeContext();

    if HaveRASExt() && SCTLR[].IESB == '1' then
        ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then                // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set s
        if PSTATE.T == '0' then
            new_pc<0> = '0';                 // T32
        else
            new_pc<1:0> = '00';              // A32
    else                                     // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc);

    if UsingAArch32() then
        // 32 most significant bits are ignored.
        BranchTo(new_pc<31:0>, BranchType_UNKNOWN);
    else
        BranchToAddr(new_pc, BranchType_ERET);
```

Library pseudocode for aarch64/instrs/countop/CountOp

```
enumeration CountOp     {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

Library pseudocode for aarch64/instrs/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =================
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```

Library pseudocode for aarch64/instrs/extendreg/ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType type, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;

    case type of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SXTX 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_UXTX 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_SXTX,
                         ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

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;
```

Library pseudocode for aarch64/instrs/integer/bitmasks/DecodeBitMasks

```
// 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 ReservedValue();
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        ReservedValue();

    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 equivalant to:
    //   esize = 1 << len;
    //   d = UInt(diff<len-1:0>);
    //   welem = ZeroExtend(Ones(S + 1), esize);
    //   telem = ZeroExtend(Ones(d + 1), esize);
    //   wmask = Replicate(ROR(welem, R));
    //   tmask = Replicate(telem);
    //   return (wmask, tmask);

    // Compute "top mask"
    tmask_and = diff<5:0> OR NOT(levels);
    tmask_or  = diff<5:0> AND levels;

    tmask = Ones(64);
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
               OR  Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
               OR  Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
               OR  Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
               OR  Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
               OR  Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
               OR  Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

    // Compute "wraparound mask"
    wmask_and = immr OR NOT(levels);
    wmask_or  = immr AND levels;

    wmask = Zeros(64);
    wmask = ((wmask
```

```
                AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
                 OR  Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
    // optimization of first step:
    // wmask = Replicate(wmask_or<0> : '0', 32);
    wmask = ((wmask
                AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
                 OR  Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
    wmask = ((wmask
                AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
                 OR  Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
    wmask = ((wmask
                AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
                 OR  Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
    wmask = ((wmask
                AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
                 OR  Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
    wmask = ((wmask
                AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
                 OR  Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

    if diff<6> != '0' then // borrow from S - R
        wmask = wmask AND tmask;
    else
        wmask = wmask OR tmask;

    return (wmask<M-1:0>, tmask<M-1:0>);
```

Library pseudocode for aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp

```
enumeration MoveWideOp  {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

Library pseudocode for aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// ===================
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && immN:imms != '1xxxxxx' then
        return FALSE;
    if sf == '0' && immN:imms != '00xxxxx' then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if S < 16 then
        // ones must not span halfword boundary when rotated
        return (-R MOD 16) <= (15 - S);

    // for MOVN must contain no more than 16 zeros
    if S >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (R MOD 16) <= (S - (width - 15));

    return FALSE;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/DecodeShift

```
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00'  return ShiftType_LSL;
        when '01'  return ShiftType_LSR;
        when '10'  return ShiftType_ASR;
        when '11'  return ShiftType_ROR;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftReg

```
// ShiftReg()
// ==========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType type, integer amount)
    bits(N) result = X[reg];
    case type of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftType

```
enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

Library pseudocode for aarch64/instrs/logicalop/LogicalOp

```
enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

Library pseudocode for aarch64/instrs/memory/memop/MemAtomicOp

```
enumeration MemAtomicOp {MemAtomicOp_ADD,
                         MemAtomicOp_BIC,
                         MemAtomicOp_EOR,
                         MemAtomicOp_ORR,
                         MemAtomicOp_SMAX,
                         MemAtomicOp_SMIN,
                         MemAtomicOp_UMAX,
                         MemAtomicOp_UMIN,
                         MemAtomicOp_SWP};
```

Library pseudocode for aarch64/instrs/memory/memop/MemOp

```
enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

Library pseudocode for aarch64/instrs/memory/prefetch/Prefetch

```
// Prefetch()
// ==========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;         // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;         // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;        // PST: prepare for store
        when '11' return;                       // unallocated hint
    target = UInt(prfop<2:1>);                  // target cache level
    stream = (prfop<0> != '0');                 // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

Library pseudocode for aarch64/instrs/system/barriers/barrierop/MemBarrierOp

```
enumeration MemBarrierOp   {MemBarrierOp_DSB, MemBarrierOp_DMB, MemBarrierOp_ISB};
```

Library pseudocode for aarch64/instrs/system/hints/syshintop/SystemHintOp

```
enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
};
```

Library pseudocode for aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_PAN, // ARMv8.1
                         PSTATEField_UAO, // ARMv8.2
                         PSTATEField_SP
                         };
```

Library pseudocode for aarch64/instrs/system/sysops/sysop/SysOp

```
// SysOp()
// =======

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;   // S1E1R
        when '100 0111 1000 000' return Sys_AT;   // S1E2R
        when '110 0111 1000 000' return Sys_AT;   // S1E3R
        when '000 0111 1000 001' return Sys_AT;   // S1E1W
        when '100 0111 1000 001' return Sys_AT;   // S1E2W
        when '110 0111 1000 001' return Sys_AT;   // S1E3W
        when '000 0111 1000 010' return Sys_AT;   // S1E0R
        when '000 0111 1000 011' return Sys_AT;   // S1E0W
        when '100 0111 1000 100' return Sys_AT;   // S12E1R
        when '100 0111 1000 101' return Sys_AT;   // S12E1W
        when '100 0111 1000 110' return Sys_AT;   // S12E0R
        when '100 0111 1000 111' return Sys_AT;   // S12E0W
        when '011 0111 0100 001' return Sys_DC;   // ZVA
        when '000 0111 0110 001' return Sys_DC;   // IVAC
        when '000 0111 0110 010' return Sys_DC;   // ISW
        when '011 0111 1010 001' return Sys_DC;   // CVAC
        when '000 0111 1010 010' return Sys_DC;   // CSW
        when '011 0111 1011 001' return Sys_DC;   // CVAU
        when '011 0111 1110 001' return Sys_DC;   // CIVAC
        when '000 0111 1110 010' return Sys_DC;   // CISW
        when '000 0111 0001 000' return Sys_IC;   // IALLUIS
        when '000 0111 0101 000' return Sys_IC;   // IALLU
        when '011 0111 0101 001' return Sys_IC;   // IVAU
        when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
        when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
        when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
        when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
        when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
        when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
        when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
        when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
        when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
        when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
        when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
        when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
        when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
        when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
        when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
        when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
        when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
        when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
        when '100 1000 0111 000' return Sys_TLBI; // ALLE2
        when '110 1000 0111 000' return Sys_TLBI; // ALLE3
        when '000 1000 0111 001' return Sys_TLBI; // VAE1
        when '100 1000 0111 001' return Sys_TLBI; // VAE2
        when '110 1000 0111 001' return Sys_TLBI; // VAE3
        when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
        when '000 1000 0111 011' return Sys_TLBI; // VAAE1
        when '100 1000 0111 100' return Sys_TLBI; // ALLE1
        when '000 1000 0111 101' return Sys_TLBI; // VALE1
        when '100 1000 0111 101' return Sys_TLBI; // VALE2
        when '110 1000 0111 101' return Sys_TLBI; // VALE3
        when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
        when '000 1000 0111 111' return Sys_TLBI; // VAALE1
    return Sys_SYS;
```

Library pseudocode for aarch64/instrs/system/sysops/sysop/SystemOp

```
enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```

Library pseudocode for aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```
enumeration VBitOp      {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

Library pseudocode for aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```
enumeration CompareOp   {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                         CompareOp_LE, CompareOp_LT};
```

Library pseudocode for aarch64/instrs/vector/crypto/enabled/CheckCryptoEnabled64

```
// CheckCryptoEnabled64()
// ======================

CheckCryptoEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
    return;
```

Library pseudocode for aarch64/instrs/vector/logical/immediateop/ImmediateOp

```
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                         ImmediateOp_ORR, ImmediateOp_BIC};
```

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/Reduce

```
// Reduce()
// ========

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize);
    lo = Reduce(op, input<half-1:0>, esize);

    case op of
        when ReduceOp_FMINNUM
            result = FPMinNum(lo, hi, FPCR);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, FPCR);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, FPCR);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, FPCR);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR);
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/ReduceOp

```
enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                      ReduceOp_FMIN, ReduceOp_FMAX,
                      ReduceOp_FADD, ReduceOp_ADD};
```

Library pseudocode for aarch64/translation/attrs/AArch64.InstructionDevice

```
// AArch64.InstructionDevice()
// ===========================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch64.InstructionDevice(AddressDescriptor addrdesc, bits(64) vaddress,
                                            bits(52) ipaddress, integer level,
                                            AccType acctype, boolean iswrite, boolean secondstage,
                                            boolean s2fs1walk)

    c = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch64.PermissionFault(ipaddress, level, acctype, iswrite,
                                                 secondstage, s2fs1walk);
    else
        addrdesc.memattrs.type = MemType_Normal;
        addrdesc.memattrs.inner.attrs = MemAttr_NC;
        addrdesc.memattrs.inner.hints = MemHint_No;
        addrdesc.memattrs.outer = addrdesc.memattrs.inner;
        addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

    return addrdesc;
```

Library pseudocode for aarch64/translation/attrs/AArch64.S1AttrDecode

```
// AArch64.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch64.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;

    mair = MAIR[];
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);

    if attrfield<7:4> == '0000' then            // Device
        memattrs.type = MemType_Device;
        case attrfield<3:0> of
            when '0000'  memattrs.device = DeviceType_nGnRnE;
            when '0100'  memattrs.device = DeviceType_nGnRE;
            when '1000'  memattrs.device = DeviceType_nGRE;
            when '1100'  memattrs.device = DeviceType_GRE;
            otherwise    Unreachable();         // Reserved, handled above

    elsif attrfield<3:0> != '0000'  then        // Normal
        memattrs.type = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';

    else
        Unreachable();                          // Reserved, handled above

    return MemAttrDefaults(memattrs);
```

Library pseudocode for aarch64/translation/attrs/AArch64.TranslateAddressS1Off

```
// AArch64.TranslateAddressS1Off()
// ===============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch64.TranslateAddressS1Off(bits(64) vaddress, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    Top = AddrTop(vaddress, FALSE, PSTATE.EL);
    if !IsZero(vaddress<Top:PAMax()>) then
        level = 0;
        ipaddress = bits(52) UNKNOWN;
        secondstage = FALSE;
        s2fs1walk = FALSE;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    default_cacheable = (HasS2Translation() && HCR_EL2.DC == '1');

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.type = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;      // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
    elsif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.type = MemType_Device;
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
    else
        // Instruction cacheability controlled by SCTLR_ELx.I
        cacheable = SCTLR[].I == '1';
        result.addrdesc.memattrs.type = MemType_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            result.addrdesc.memattrs.inner.hints = MemHint_No;
        result.addrdesc.memattrs.shareable = TRUE;
        result.addrdesc.memattrs.outershareable = TRUE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.physicaladdress = vaddress<51:0>;
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch64.NoFault();
    return result;
```

Library pseudocode for aarch64/translation/checks/AArch64.AccessIsPrivileged

```
// AArch64.AccessIsPrivileged()
// ============================

boolean AArch64.AccessIsPrivileged(AccType acctype)

    if PSTATE.EL == EL0 then
        ispriv = FALSE;
    elsif PSTATE.EL == EL3 then
        ispriv = TRUE;
    elsif PSTATE.EL == EL2 && (!IsInHost() || HCR_EL2.TGE == '0') then
        ispriv = TRUE;
    elsif HaveUAOExt() && PSTATE.UAO == '1' then
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType_UNPRIV);

    return ispriv;
```

Library pseudocode for aarch64/translation/checks/AArch64.CheckPermission

```
// AArch64.CheckPermission()
// =========================
// Function used for permission checking from AArch64 stage 1 translations

FaultRecord AArch64.CheckPermission(Permissions perms, bits(64) vaddress, integer level,
                                    bit NS, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    wxn = SCTLR[].WXN == '1';

    if PSTATE.EL IN {EL0,EL1} || IsInHost() then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';

        ispriv = AArch64.AccessIsPrivileged(acctype);

        pan = if HavePANExt() then PSTATE.PAN else '0';
        if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1 && !IsSecure() && PSTATE.
            pan = '0';
        if (pan == '1' && user_r && ispriv &&
            !(acctype IN {AccType_DC,AccType_AT,AccType_IFETCH}) ||
            (acctype == AccType_AT && AArch64.ExecutingATS1xPInstr())) then
            priv_r = FALSE;
            priv_w = FALSE;

        user_xn = perms.xn == '1' || (user_w && wxn);
        priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2 or EL3
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
        xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW } then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(52) UNKNOWN;
        return AArch64.PermissionFault(ipaddress, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

Library pseudocode for aarch64/translation/checks/AArch64.CheckS2Permission

```
// AArch64.CheckS2Permission()
// ===========================
// Function used for permission checking from AArch64 stage 2 translations

FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, bits(52) ipaddress,
                                      integer level, AccType acctype, boolean iswrite,
                                      boolean s2fs1walk, boolean hwupdatewalk)

    assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HasS2Translation();

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    if HaveExtendedExecuteNeverExt() then
        case perms.xn:perms.xxn of
            when '00'  xn = FALSE;
            when '01'  xn = PSTATE.EL == EL1;
            when '10'  xn = TRUE;
            when '11'  xn = PSTATE.EL == EL0;
    else
        xn = perms.xn == '1';
    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
        failedread = TRUE;
    elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW }) && !s2fs1walk then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    elsif hwupdatewalk then
        fail = !w;
        failedread = !iswrite;
    else
        fail = !r;
        failedread = !iswrite;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch64.PermissionFault(ipaddress, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, 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, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch64.NoFault();

    d_side = (acctype != AccType_IFETCH);
    generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch64.CheckBreakpoint(vaddress, size);

    return fault;
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = AArch64.AccessIsPrivileged(acctype);

    for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
        match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

Library pseudocode for aarch64/translation/faults/AArch64.AccessFlagFault

```
// AArch64.AccessFlagFault()
// =========================

FaultRecord AArch64.AccessFlagFault(bits(52) ipaddress, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.AddressSizeFault

```
// AArch64.AddressSizeFault()
// ==========================

FaultRecord AArch64.AddressSizeFault(bits(52) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.AlignmentFault

```
// AArch64.AlignmentFault()
// ========================

FaultRecord AArch64.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch64.CreateFaultRecord(Fault_Alignment, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.AsynchExternalAbort

```
// AArch64.AsynchExternalAbort()
// ============================
// Wrapper function for asynchronous external aborts

FaultRecord AArch64.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

    type = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(type, ipaddress,  level, acctype, iswrite, extflag,
                                     errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// ====================

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)

    ipaddress = bits(52) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_Debug, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.NoFault

```
// AArch64.NoFault()
// =================

FaultRecord AArch64.NoFault()

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_None, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.PermissionFault

```
// AArch64.PermissionFault()
// =========================

FaultRecord AArch64.PermissionFault(bits(52) ipaddress, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Permission, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.TranslationFault

```
// AArch64.TranslationFault()
// ==========================

FaultRecord AArch64.TranslationFault(bits(52) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Translation, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/translation/AArch64.CheckAndUpdateDescriptor

```
// AArch64.CheckAndUpdateDescriptor()
// ==================================
// Check and update translation table descriptor if hardware update is configured

FaultRecord AArch64.CheckAndUpdateDescriptor(DescriptorUpdate result, FaultRecord fault,
                                             boolean secondstage, bits(64) vaddress, AccType acctype,
                                             boolean iswrite, boolean s2fs1walk, boolean hwupdatewalk)

    // Check if access flag can be updated
    // Address translation instructions are permitted to update AF but not required
    if result.AF then
        if fault.type == Fault_None then
            hw_update_AF = TRUE;
        elsif ConstrainUnpredictable(Unpredictable_AFUPDATE) == Constraint_TRUE then
            hw_update_AF = TRUE;
        else
            hw_update_AF = FALSE;

    if result.AP && fault.type == Fault_None then
        write_perm_req = (iswrite || acctype IN {AccType_ATOMICRW,AccType_ORDEREDRW}) && !s2fs1walk;
        hw_update_AP = (write_perm_req && !(acctype IN {AccType_AT,AccType_DC})) || hwupdatewalk;
    else
        hw_update_AP = FALSE;

    if hw_update_AF || hw_update_AP then
        if secondstage || !HasS2Translation() then
            descaddr2 = result.descaddr;
        else
            hwupdatewalk = TRUE;
            descaddr2 = AArch64.SecondStageWalk(result.descaddr, vaddress, acctype, iswrite, 8, hwupdat
            if IsFault(descaddr2) then
                return descaddr2.fault;

        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        desc = _Mem[descaddr2, 8, accdesc];

        if hw_update_AF then
            desc<10> = '1';
        if hw_update_AP then
            desc<7> = (if secondstage then '1' else '0');

        _Mem[descaddr2,8,accdesc] = desc;

    return fault;
```

Library pseudocode for aarch64/translation/translation/AArch64.FirstStageTranslate

```
// AArch64.FirstStageTranslate()
// =============================
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.FirstStageTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                              boolean wasaligned, integer size)

    if HasS2Translation() then
        s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';
    else
        s1_enabled = SCTLR[].M == '1';

    ipaddress = bits(52) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                          // First stage enabled
        S1 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);
        permissioncheck = TRUE;
    else
        S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;

        if UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() then
            if S1.addrdesc.memattrs.type == MemType_Device && S1.addrdesc.memattrs.device != DeviceType
                nTLSMD = if S1TranslationRegime() == EL2 then SCTLR_EL2.nTLSMD else SCTLR_EL1.nTLSMD;
                if nTLSMD == '0' then
                    S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
    // Check for unaligned data accesses to Device memory
    if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
        && S1.addrdesc.memattrs.type == MemType_Device && !IsFault(S1.addrdesc) then
        S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level,
                                                    S1.addrdesc.paddress.NS,
                                                    acctype, iswrite);

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
        acctype == AccType_IFETCH) then
        S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                                acctype, iswrite,
                                                secondstage, s2fs1walk);

    // Check and update translation table descriptor if required
    hwupdatewalk = FALSE;
    s2fs1walk = FALSE;
    S1.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S1.descupdate, S1.addrdesc.fault,
                                                         secondstage, vaddress, acctype,
                                                         iswrite, s2fs1walk, hwupdatewalk);

    return S1.addrdesc;
```

Library pseudocode for aarch64/translation/translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch64.FullTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch64.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(S1) && HasS2Translation() then
        s2fs1walk = FALSE;
        hwupdatewalk = FALSE;
        result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                              size, hwupdatewalk);
    else
        result = S1;

    return result;
```

Library pseudocode for aarch64/translation/translation/AArch64.SecondStageTranslate

```
// AArch64.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.SecondStageTranslate(AddressDescriptor S1, bits(64) vaddress,
                                               AccType acctype, boolean iswrite, boolean wasaligned,
                                               boolean s2fs1walk, integer size, boolean hwupdatewalk)
    assert HasS2Translation();

    s2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';
    secondstage = TRUE;

    if s2_enabled then                        // Second stage enabled
        ipaddress = S1.paddress.physicaladdress<51:0>;

        S2 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
            && S2.addrdesc.memattrs.type == MemType_Device && !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                          acctype, iswrite, s2fs1walk, hwupdatewalk);
        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype == AccType_IFETCH) then
            S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    acctype, iswrite,
                                                    secondstage, s2fs1walk);

        // Check for protected table walk
        if (s2fs1walk && !IsFault(S2.addrdesc) && HCR_EL2.PTW == '1' &&
            S2.addrdesc.memattrs.type == MemType_Device) then
            S2.addrdesc.fault = AArch64.PermissionFault(ipaddress, S2.level, acctype,
                                                        iswrite, secondstage, s2fs1walk);

        // Check and update translation table descriptor if required
        S2.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S2.descupdate, S2.addrdesc.fault,
                                                             secondstage, vaddress, acctype,
                                                             iswrite, s2fs1walk, hwupdatewalk);
        result = CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

    return result;
```

Library pseudocode for aarch64/translation/translation/AArch64.SecondStageWalk

```
// AArch64.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype,
                                          boolean iswrite, integer size, boolean hwupdatewalk)

    assert HasS2Translation();

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                        size, hwupdatewalk);
```

Library pseudocode for aarch64/translation/translation/AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)

    result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
```

Library pseudocode for aarch64/translation/translation/IsEL1TransRegimeRegs

```
// IsEL1TransRegimeRegs()
// ======================

// Returns TRUE if its a register in EL1 translation regime

boolean IsEL1TransRegimeRegs()
    return !HaveEL(EL2) || PSTATE.EL == EL1 || (PSTATE.EL == EL0 && (HCR_EL2.E2H == '0' ||  HCR_EL2.TGE
```

Library pseudocode for aarch64/translation/walk/AArch64.TranslationTableWalk

```
// AArch64.TranslationTableWalk()
// ==============================
// Returns a result of a translation table walk
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch64.TranslationTableWalk(bits(52) ipaddress, bits(64) vaddress,
                                       AccType acctype, boolean iswrite, boolean secondstage,
                                       boolean s2fs1walk, integer size)
    if !secondstage then
        assert !ELUsingAArch32(S1TranslationRegime());
    else
        assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HasS2Translation();

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(64) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2

    descaddr.memattrs.type = MemType_Normal;

    // Derived parameters for the page table walk:
    //  grainsize = Log2(Size of Table)         - Size of Table is 4KB, 16KB or 64KB in AArch64
    //  stride = Log2(Address per Level)        - Bits of address consumed at each level
    //  firstblocklevel = First level where a block entry is allowed
    //  ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTCR_EL2.PS
    //  inputsize = Log2(Size of Input Address) - Input Address size in bits
    //  level = Level to start walk from
    // This means that the number of levels after start level = 3-level

    if !secondstage then
        // First stage translation
        inputaddr = ZeroExtend(vaddress);
        top = AddrTop(inputaddr, (acctype == AccType_IFETCH), PSTATE.EL);
        if PSTATE.EL == EL3 then
            largegrain = TCR_EL3.TG0 == '01';
            midgrain = TCR_EL3.TG0 == '10';
            inputsize = 64 - UInt(TCR_EL3.T0SZ);
            inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
            if inputsize > inputsize_max then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - 39;
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            ps = TCR_EL3.PS;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<to
            disabled = FALSE;
            baseregister = TTBR0_EL3;
            descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0, secondstage);
            reversedescriptors = SCTLR_EL3.EE == '1';
            lookupsecure = TRUE;
            singlepriv = TRUE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL3.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL3.HD == '1';
            hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL3.HPD == '1';
        elsif IsInHost() then
            if inputaddr<top> == '0' then
                largegrain = TCR_EL2.TG0 == '01';
                midgrain = TCR_EL2.TG0 == '10';
                inputsize = 64 - UInt(TCR_EL2.T0SZ);
                inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
                if inputsize > inputsize_max then
                    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
```

```
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - 39;
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputadd
            disabled = TCR_EL2.EPD0 == '1';
            baseregister = TTBR0_EL2;
            descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondsta
            hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD0 == '1';
        else
            inputsize = 64 - UInt(TCR_EL2.T1SZ);
            largegrain = TCR_EL2.TG1 == '11';       // TG1 and TG0 encodings differ
            midgrain = TCR_EL2.TG1 == '01';
            inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
            if inputsize > inputsize_max then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - 39;
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputadd
            disabled = TCR_EL2.EPD1 == '1';
            baseregister = TTBR1_EL2;
            descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH1, TCR_EL2.ORGN1, TCR_EL2.IRGN1, secondsta
            hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD1 == '1';
        ps = TCR_EL2.IPS;
        reversedescriptors = SCTLR_EL2.EE == '1';
        lookupsecure = FALSE;
        singlepriv = FALSE;
        update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
    elsif PSTATE.EL == EL2 then
        inputsize = 64 - UInt(TCR_EL2.T0SZ);
        largegrain = TCR_EL2.TG0 == '01';
        midgrain = TCR_EL2.TG0 == '10';
        inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
        if inputsize > inputsize_max then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_max;
        inputsize_min = 64 - 39;
        if inputsize < inputsize_min then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_min;
        ps = TCR_EL2.PS;
        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<to
        disabled = FALSE;
        baseregister = TTBR0_EL2;
        descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondstage);
        reversedescriptors = SCTLR_EL2.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;
        update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
        hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD == '1';
    else
        if inputaddr<top> == '0' then
            inputsize = 64 - UInt(TCR_EL1.T0SZ);
            largegrain = TCR_EL1.TG0 == '01';
            midgrain = TCR_EL1.TG0 == '10';
            inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
            if inputsize > inputsize_max then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
```

```
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - 39;
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputadd
            disabled = TCR_EL1.EPD0 == '1';
            baseregister = TTBR0_EL1;
            descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORGN0, TCR_EL1.IRGN0, secondsta
            hierattrsdisabled = AArch64.HaveHPDExt() &&  TCR_EL1.HPD0 == '1';
        else
            inputsize = 64 - UInt(TCR_EL1.T1SZ);
            largegrain = TCR_EL1.TG1 == '11';       // TG1 and TG0 encodings differ
            midgrain = TCR_EL1.TG1 == '01';
            inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
            if inputsize > inputsize_max then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - 39;
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputadd
            disabled = TCR_EL1.EPD1 == '1';
            baseregister = TTBR1_EL1;
            descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGN1, TCR_EL1.IRGN1, secondsta
            hierattrsdisabled = AArch64.HaveHPDExt() &&  TCR_EL1.HPD1 == '1';
        ps = TCR_EL1.IPS;
        reversedescriptors = SCTLR_EL1.EE == '1';
        lookupsecure = IsSecure();
        singlepriv = FALSE;
        update_AF = HaveAccessFlagUpdateExt() && TCR_EL1.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL1.HD == '1';
    if largegrain then
        grainsize = 16;                                         // Log2(64KB page size)
        firstblocklevel = (if Have52BitPAExt() then 1 else 2);  // Largest block is 4TB (2^42 b
                                                                // and 512MB (2^29 bytes) other
    elsif midgrain then
        grainsize = 14;                                         // Log2(16KB page size)
        firstblocklevel = 2;                                    // Largest block is 32MB (2^25
    else // Small grain
        grainsize = 12;                                         // Log2(4KB page size)
        firstblocklevel = 1;                                    // Largest block is 1GB (2^30 b
    stride = grainsize - 3;                                     // Log2(page size / 8 bytes)
    // The starting level is the number of strides needed to consume the input address
    level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));

else
    // Second stage translation
    inputaddr = ZeroExtend(ipaddress);
    inputsize = 64 - UInt(VTCR_EL2.T0SZ);
    largegrain = VTCR_EL2.TG0 == '01';
    midgrain = VTCR_EL2.TG0 == '10';
    inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
    if inputsize > inputsize_max then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_max;
    inputsize_min = 64 - 39;
    if inputsize < inputsize_min then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_min;
    ps = VTCR_EL2.PS;
    basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<63:inp
    disabled = FALSE;
    baseregister = VTTBR_EL2;
```

```
    descaddr.memattrs = WalkAttrDecode(VTCR_EL2.IRGN0, VTCR_EL2.ORGN0, VTCR_EL2.SH0, secondstage);
    reversedescriptors = SCTLR_EL2.EE == '1';
    lookupsecure = FALSE;
    singlepriv = TRUE;
    update_AF = HaveAccessFlagUpdateExt() && VTCR_EL2.HA == '1';
    update_AP = HaveDirtyBitModifierExt() && update_AF && VTCR_EL2.HD == '1';

    startlevel = UInt(VTCR_EL2.SL0);
    if largegrain then
        grainsize = 16;                                         // Log2(64KB page size)
        level = 3 - startlevel;
        firstblocklevel = (if Have52BitPAExt() then 1 else 2);  // Largest block is 4TB (2^42 bytes
                                                                // and 512MB (2^29 bytes) otherwise
    elsif midgrain then
        grainsize = 14;                                         // Log2(16KB page size)
        level = 3 - startlevel;
        firstblocklevel = 2;                                    // Largest block is 32MB (2^25 byte
    else // Small grain
        grainsize = 12;                                         // Log2(4KB page size)
        level = 2 - startlevel;
        firstblocklevel = 1;                                    // Largest block is 1GB (2^30 bytes
    stride = grainsize - 3;                                     // Log2(page size / 8 bytes)

    // Limits on IPA controls based on implemented PA size. Level 0 is only
    // supported by small grain translations
    if largegrain then                              // 64KB pages
        // Level 1 only supported if implemented PA size is greater than 2^42 bytes
        if level == 0 || (level == 1 && PAMax() <= 42) then basefound = FALSE;
    elsif midgrain then                             // 16KB pages
        // Level 1 only supported if implemented PA size is greater than 2^40 bytes
        if level == 0 || (level == 1 && PAMax() <= 40) then basefound = FALSE;
    else                                            // Small grain, 4KB pages
        // Level 0 only supported if implemented PA size is greater than 2^42 bytes
        if level < 0 || (level == 0 && PAMax() <= 42) then basefound = FALSE;

    // If the inputsize exceeds the PAMax value, the behavior is CONSTRAINED UNPREDICTABLE
    inputsizecheck = inputsize;
    if inputsize > PAMax() && (!ELUsingAArch32(EL1) || inputsize > 40) then
        case ConstrainUnpredictable(Unpredictable_LARGEIPA) of
            when Constraint_FORCE
                // Restrict the inputsize to the PAMax value
                inputsize = PAMax();
                inputsizecheck = PAMax();
            when Constraint_FORCENOSLCHECK
                // As FORCE, except use the configured inputsize in the size checks below
                inputsize = PAMax();
            when Constraint_FAULT
                // Generate a translation fault
                basefound = FALSE;
            otherwise
                Unreachable();

    // Number of entries in the starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    startsizecheck = inputsizecheck - ((3 - level)*stride + grainsize); // Log2(Num of entries)

    // Check for starting level table with fewer than 2 entries or longer than 16 pages.
    // Lower bound check is:  startsizecheck < Log2(2 entries)
    // Upper bound check is:  startsizecheck > Log2(pagesize/8*16)
    if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

if !basefound || disabled then
    level = 0;            // AArch32 reports this as a level 1 fault
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

case ps of
    when '000'  outputsize = 32;
    when '001'  outputsize = 36;
```

```
        when '010'  outputsize = 40;
        when '011'  outputsize = 42;
        when '100'  outputsize = 44;
        when '101'  outputsize = 48;
        when '110'  outputsize = (if Have52BitPAExt() && largegrain then 52 else 48);
        otherwise   outputsize = 48;

    if outputsize > PAMax() then outputsize = PAMax();

    if outputsize < 48 && !IsZero(baseregister<47:outputsize>) then
        level = 0;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    // Bottom bound of the Base address is:
    //     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
    // Number of entries in starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
    if outputsize == 52 then
        z = (if baselowerbound < 6 then 6 else baselowerbound);
        baseaddress = baseregister<5:2>:baseregister<47:z>:Zeros(z);
    else
        baseaddress = ZeroExtend(baseregister<47:baselowerbound>:Zeros(baselowerbound));

    ns_table = if lookupsecure then '0' else '1';
    ap_table = '00';
    xn_table = '0';
    pxn_table = '0';

    addrselecttop = inputsize - 1;

    apply_nvnv1_effect = HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1;

    repeat
        addrselectbottom = (3-level)*stride + grainsize;

        bits(52) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
        descaddr.paddress.physicaladdress = baseaddress OR index;
        descaddr.paddress.NS = ns_table;

        // If there are two stages of translation, then the first stage table walk addresses
        // are themselves subject to translation
        if secondstage || !HasS2Translation() then
            descaddr2 = descaddr;
        else
            hwupdatewalk = FALSE;
            descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
            // Check for a fault on the stage 2 walk
            if IsFault(descaddr2) then
                result.addrdesc.fault = descaddr2.fault;
                return result;

        // Update virtual address for abort functions
        descaddr2.vaddress = ZeroExtend(vaddress);

        accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
        desc = _Mem[descaddr2, 8, accdesc];

        if reversedescriptors then desc = BigEndianReverse(desc);

        if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
             // Fault (00), Reserved (10), or Block (01) at level 3
            result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        // Valid Block, Page, or Table entry
        if desc<1:0> == '01' || level == 3 then                 // Block (01) or Page (11)
```

```
            blocktranslate = TRUE;
        else                                                        // Table (11)
            if (outputsize < 52 && largegrain && !IsZero(desc<15:12>)) || (outputsize < 48 && !IsZero(d
                result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                                  iswrite, secondstage, s2fs1walk);
                return result;

            if outputsize == 52 then
                baseaddress = desc<15:12>:desc<47:grainsize>:Zeros(grainsize);
            else
                baseaddress = ZeroExtend(desc<47:grainsize>:Zeros(grainsize));
            if !secondstage then
                // Unpack the upper and lower table attributes
                ns_table    = ns_table    OR desc<63>;
            if !secondstage && !hierattrsdisabled then
                ap_table<1> = ap_table<1> OR desc<62>;       // read-only

                if apply_nvnv1_effect then
                    pxn_table   = pxn_table   OR desc<60>;
                else
                    xn_table    = xn_table    OR desc<60>;
                // pxn_table and ap_table[0] apply in EL1&0 or EL2&0 translation regimes
                if !singlepriv then
                    if !apply_nvnv1_effect then
                        pxn_table   = pxn_table   OR desc<59>;
                        ap_table<0> = ap_table<0> OR desc<61>;   // privileged

            level = level + 1;
            addrselecttop = addrselectbottom - 1;
            blocktranslate = FALSE;
    until blocktranslate;

    // Check block size is supported at this level
    if level < firstblocklevel then
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Check for misprogramming of the contiguous bit
    if largegrain then
        contiguousbitcheck = level == 2 && inputsize < 34;
    elsif midgrain then
        contiguousbitcheck = level == 2 && inputsize < 30;
    else
        contiguousbitcheck = level == 1 && inputsize < 34;

    if contiguousbitcheck && desc<52> == '1' then
        if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
            result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

    // Check the output address is inside the supported range
    if (outputsize < 52 && largegrain && !IsZero(desc<15:12>)) || (outputsize < 48 && !IsZero(desc<47:c
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Unpack the descriptor into address and upper and lower block attributes
    if outputsize == 52 then
        outputaddress = desc<15:12>:desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
    else
        outputaddress = ZeroExtend(desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>);
    // Check Access Flag
    if desc<10> == '0' then
        if !update_AF then
            result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress, level, acctype,
                                                            iswrite, secondstage, s2fs1walk);
            return result;
        else
```

```
            result.descupdate.AF = TRUE;

    if update_AP && desc<51> == '1' then
        // If hw update of access permission field is configured consider AP[2] as '0' / S2AP[2] as '1'
        if !secondstage && desc<7> == '1' then
            desc<7> = '0';
            result.descupdate.AP = TRUE;
        elsif secondstage && desc<7> == '0' then
            desc<7> = '1';
            result.descupdate.AP = TRUE;

    // Required descriptor if AF or AP[2]/S2AP[2] needs update
    result.descupdate.descaddr = descaddr;

    if apply_nvnv1_effect then
        pxn = desc<54>;
        xn = '0';
    else
        xn = desc<54>;
        pxn = desc<53>;
    contiguousbit = desc<52>;
    nG = desc<11>;
    sh = desc<9:8>;
    if apply_nvnv1_effect then
        ap = desc<7>:'01';                                  // Bit[6] is treated as 0 regardless of v
    else
        ap = desc<7:6>:'1';
    memattr = desc<5:2>;                                    // AttrIndx and NS bit in stage 1

    result.domain = bits(4) UNKNOWN;                        // Domains not used
    result.level = level;
    result.blocksize = 2^((3-level)*stride + grainsize);

    // Stage 1 translation regimes also inherit attributes from the tables
    if !secondstage then
        result.perms.xn      = xn OR xn_table;
        result.perms.ap<2>   = ap<2> OR ap_table<1>;        // Force read-only
        // PXN, nG and AP[1] apply in EL1&0 or EL2&0 stage 1 translation regimes
        if !singlepriv then
            result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
            result.perms.pxn   = pxn OR pxn_table;
            // Pages from Non-secure tables are marked non-global in Secure EL1&0
            if IsSecure() then
                result.nG = nG OR ns_table;
            else
                result.nG = nG;
        else
            result.perms.ap<1> = '1';
            result.perms.pxn   = '0';
            result.nG          = '0';
        result.perms.ap<0>   = '1';
        result.addrdesc.memattrs = AArch64.S1AttrDecode(sh, memattr<2:0>, acctype);
        result.addrdesc.paddress.NS = memattr<3> OR ns_table;
    else
        result.perms.ap<2:1> = ap<2:1>;
        result.perms.ap<0>   = '1';
        result.perms.xn      = xn;
        if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
        result.perms.pxn     = '0';
        result.nG            = '0';
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
        result.addrdesc.paddress.NS = '1';

    result.addrdesc.paddress.physicaladdress = outputaddress;
    result.addrdesc.fault = AArch64.NoFault();
    result.contiguous = contiguousbit == '1';
    if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;
    return result;
```

Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';            // Clear TX underrun flag
    EDSCR.RXO = '0';            // Clear RX overrun flag

    if Halted() then            // in Debug state
        EDSCR.ITO = '0';        // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO) then
        return;
    EDSCR.ERR = '0';            // Clear cumulative error flag

    return;
```

Library pseudocode for shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =================

bits(2) DebugTargetFrom(boolean secure)
    if HaveEL(EL2) && !secure then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns the status of EDPRSR.EDAD.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if ExternalInvasiveDebugEnabled() then
        if ExternalSecureInvasiveDebugEnabled() then
            return TRUE;
        elsif HaveEL(EL3) then
            return (if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD) == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// ========================
// Returns the status of EDPRSR.EPMAD.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if ExternalNoninvasiveDebugEnabled() then
        if ExternalSecureNoninvasiveDebugEnabled() then
            return TRUE;
        elsif HaveEL(EL3) then
            return (if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD) == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

Library pseudocode for shared/debug/authentication/Debug_authentication

```
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
```

Library pseudocode for shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================

boolean ExternalInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, ExternalInvasiveDebugEnabled returns the state of the DBGEN
    // signal.
    return DBGEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =================================

boolean ExternalNoninvasiveDebugAllowed()
    // Return TRUE if Trace and PC Sample-based Profiling are allowed
    return (ExternalNoninvasiveDebugEnabled() &&
            (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled() ||
             (ELUsingAArch32(EL1) && PSTATE.EL == EL0 && SDER.SUNIDEN == '1')));
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =================================

boolean ExternalNoninvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN
    // OR NIDEN) signal.
    return ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================

boolean ExternalSecureInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, ExternalSecureInvasiveDebugEnabled returns the state of the
    // (DBGEN AND SPIDEN) signal.
    // CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================

boolean ExternalSecureNoninvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, ExternalSecureNoninvasiveDebugEnabled returns the state of the
    // (DBGEN OR NIDEN) AND (SPIDEN OR SPNIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
```

Library pseudocode for shared/debug/cti/CTI_SetEventLevel

```
// Set a Cross Trigger multi-cycle input event trigger to the specified level.
CTI_SetEventLevel(CrossTriggerIn id, signal level);
```

Library pseudocode for shared/debug/cti/CTI_SignalEvent

```
// Signal a discrete event on a Cross Trigger input event trigger.
CTI_SignalEvent(CrossTriggerIn id);
```

Library pseudocode for shared/debug/cti/CrossTrigger

```
enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,  CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,  CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,     CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,         CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,  CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,  CrossTriggerIn_TraceExtOut3};
```

Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    if ELUsingAArch32(EL1) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                   (commtx && DBGDCCINT.TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                   (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRRX_EL0

```
// DBGDTRRX_EL0[] (external write)
// ==============================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xD5330501<31:0>);               // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401<31:0>);               // A64 "STR W1,[X0],#4"
            X[1] = bits(64) UNKNOWN;
        else
            ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
            R[1] = bits(32) UNKNOWN;
        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(32) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
            assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_EL0[] (external read)
// ==============================

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRTX_EL0

```
// DBGDTRTX_EL0[] (external read)
// ==============================
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then           // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(32) UNKNOWN;
        else
            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';

        if !UsingAArch32() then
            X[1] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// ===============================

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write
    DTRTX = value;
    return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTR_EL0

```
// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ===================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```

Library pseudocode for shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

Library pseudocode for shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if !Halted() then return;                           // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';              // Overrun condition: block write
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```

Library pseudocode for shared/debug/halting/DCPSInstruction

```
// 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 HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then UndefinedFault();
            else handle_el = EL1;

        when EL2
            if !HaveEL(EL2) then UndefinedFault();
            elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
            elsif IsSecure() then UndefinedFault();
            else handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then UndefinedFault();
            handle_el = EL3;
        otherwise
            Unreachable();

    from_secure = IsSecure();
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32();                  // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1
                AArch32.WriteMode(M32_Svc);
                if HavePANExt() && SCTLR.SPAN == '0' then
                    PSTATE.PAN = '1';
            when EL2  AArch32.WriteMode(M32_Hyp);
            when EL3
                AArch32.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();
        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(32) UNKNOWN;  ESR[] = bits(32) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;

        if HaveUAOExt() then PSTATE.UAO = '0';

    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    // SCTLR[].IESB might be ignored in Debug state.
    if HaveRASExt() && SCTLR[].IESB == '1' && ConstrainUnpredictableBool(Unpredictable_IESBinDebug) the
        ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);
    return;
```

Library pseudocode for shared/debug/halting/DRPSInstruction

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    // SCTLR[].IESB might be ignored in Debug state.
    if HaveRASExt() && SCTLR[].IESB == '1' && ConstrainUnpredictableBool(Unpredictable_IESBinDebug) the
        ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);

    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(32) UNKNOWN;

    UpdateEDSCRFields();                                // Update EDSCR PE state flags.

    return;
```

Library pseudocode for shared/debug/halting/DebugHalt

```
constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```
DisableITRAndResumeInstructionPrefetch();
```

Library pseudocode for shared/debug/halting/ExecuteA64

```
// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);
```

Library pseudocode for shared/debug/halting/ExecuteT32

```
// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);
```

Library pseudocode for shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// ================

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001';                          // Signal restarting
    EDESR<2:0> = '000';                               // Clear any pending Halting debug events

    bits(64) new_pc;
    bits(32) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR);
        spsr = DSPSR;
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr);                           // Can update privileged bits, even at EL0.

    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
        BranchTo(new_pc<31:0>, BranchType_UNKNOWN);    // AArch32 branch
    else
        // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
        if spsr<4> == '1' && ConstrainUnpredictableBool(Unpredictable_RESTARTZEROUPPERPC) then
            new_pc<63:32> = Zeros();
        BranchTo(new_pc, BranchType_DBGEXIT);         // A type of branch that is never predicted

    (EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');       // Atomically signal restarted
    UpdateEDSCRFields();                              // Stop signalling PE state.
    DisableITRAndResumeInstructionPrefetch();

    return;
```

Library pseudocode for shared/debug/halting/Halt

```
// Halt()
// ======

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt
    if UsingAArch32() then
        DLR = ThisInstrAddr();
        DSPSR = GetPSRFromPSTATE();
        DSPSR.SS = PSTATE.SS;                   // Always save PSTATE.SS
    else
        DLR_EL0 = ThisInstrAddr();
        DSPSR_EL0 = GetPSRFromPSTATE();
        DSPSR_EL0.SS = PSTATE.SS;               // Always save PSTATE.SS

    EDSCR.ITE = '1';  EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0';                        // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = (if ExternalSecureInvasiveDebugEnabled() then '0' else '1');
    else
        assert EDSCR.SDD == '1';                // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';
    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if
    // UNKNOWN. PSTATE.{N,Z,C,V,Q,GE} are also not observable, but since these are not changed on
    // exception entry, this function also leaves them unchanged. PSTATE.{E,M,nRW,EL,SP} are
    // unchanged. PSTATE.IL is set to 0.
    if UsingAArch32() then
        PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1'; // PSTATE.J is RES0
    else
        PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                      // Signal entered Debug state
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    return;
```

Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

Library pseudocode for shared/debug/halting/Halted

```
// Halted()
// ========

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});                    // Halted
```

Library pseudocode for shared/debug/halting/HaltingAllowed

```
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elsif 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') 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.
    base = if IsSecure() then 0 else 4;
    if HaltingAllowed() then
        if HaveExtendedECDebugEvents() then
            exception_exit = !exception_entry;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + 8>:EDECCR<UInt(PSTATE.EL) + base>;
            case ctrl of
                when '00'  halt = FALSE;
                when '01'  halt = TRUE;
                when '10'  halt = (exception_exit == TRUE);
                when '11'  halt = (exception_entry == TRUE);
        else
            halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');
        if halt then Halt(DebugHalt_ExceptionCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
```

Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if EDECR.OSUCE == '1' && !Halted() then EDESR.OSUC = '1';
```

Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if EDECR.RCE == '1' then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt(DebugHalt_EDBGRQ);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();
```

Library pseudocode for shared/debug/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// ================

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exeception.
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted();             // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();

    if active && reset then                     // Coming out of reset with EDECR.SS set.
        EDESR.SS = '1';
    elsif active && HaltingAllowed() then
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    case target of
        when EL3
            int_dis = (EDSCR.INTdis == '11' && 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_COMMIRQ, InterruptID_CTIIRQ,
                         InterruptID_COMMRX, InterruptID_COMMTX};
```

Library pseudocode for shared/debug/interrupts/SetInterruptRequestLevel

```
// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);
```

Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// ================

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.

    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ns = if IsSecure() then '0' else '1';
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1;
    if HaveEL(EL2) && !IsSecure() 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() && !IsSecure() && !ELUsingAArch32(EL2) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
    return;
```

Library pseudocode for shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// =================

bits(32) EDPCSRlo[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || EDLSR.SLK == '0';        // Software locked: no side-effects

    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDPCSRhi.PC  = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
                EDPCSRhi.EL  = pc_sample.el;
                EDPCSRhi.NS  = pc_sample.ns;
            else
                EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDVIDSR = pc_sample.contextidr_el2;
            else
                EDVIDSR.VMID = (if HaveEL(EL2) && pc_sample.ns == '1' && pc_sample.el IN {EL1,EL0}
                                then pc_sample.vmid else Zeros(16));
                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,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    bits(16) vmid
)

PCSample pc_sample;
```

Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    if !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() then
        if MDSCR_EL1.SS == '1' && PSTATE.SS == '0' then
            AArch64.SoftwareStepException();
```

Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(32) spsr)
    assert Halted() || Restarting() ||  PSTATE.EL != EL0;

    SS_bit = '0';

    if MDSCR_EL1.SS == '1' then
        if Restarting() then
            enabled_at_source = FALSE;
        elsif UsingAArch32() then
            enabled_at_source = AArch32.GenerateDebugExceptions();
        else
            enabled_at_source = AArch64.GenerateDebugExceptions();

        if IllegalExceptionReturn(spsr) then
            dest = PSTATE.EL;
        else
            (valid, dest) = ELFromSPSR(spsr);  assert valid;

        secure = IsSecureBelowEL3() || dest == EL3;

        if ELUsingAArch32(dest) then
            enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
        else
            mask = spsr<9>;
            enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);

        ELd = DebugTargetFrom(secure);
        if !ELUsingAArch32(ELd) && !enabled_at_source && enabled_at_dest then
            SS_bit = spsr<21>;
    return SS_bit;
```

Library pseudocode for shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean SoftwareStep_DidNotStep();
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean SoftwareStep_SteppedEX();
```

Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// ===================
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    if UsingAArch32() then
        cond = AArch32.CurrentCond();
        if PSTATE.T == '0' then             // A32
            syndrome<4> = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else                                // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            //  * CV set to 0 and COND is set to an UNKNOWN value
            //  * CV set to 1 and COND is set to the condition code for the condition that
            //    applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';

    return syndrome;
```

Library pseudocode for shared/exceptions/exceptions/Exception

```
enumeration Exception {Exception_Uncategorized,       // Uncategorized or unknown reason
                       Exception_WFxTrap,             // Trapped WFI or WFE instruction
                       Exception_CP15RTTrap,          // Trapped AArch32 MCR or MRC access to CP15
                       Exception_CP15RRTTrap,         // Trapped AArch32 MCRR or MRRC access to CP15
                       Exception_CP14RTTrap,          // Trapped AArch32 MCR or MRC access to CP14
                       Exception_CP14DTTrap,          // Trapped AArch32 LDC or STC access to CP14
                       Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
                       Exception_FPIDTrap,            // Trapped access to SIMD or FP ID register
                       // Trapped BXJ instruction not supported in ARMv8
                       Exception_PACTrap,             // Trapped invalid PAC use
                       Exception_CP14RRTTrap,         // Trapped MRRC access to CP14 from AArch32
                       Exception_IllegalState,        // Illegal Execution state
                       Exception_SupervisorCall,      // Supervisor Call
                       Exception_HypervisorCall,      // Hypervisor Call
                       Exception_MonitorCall,         // Monitor Call or Trapped SMC instruction
                       Exception_SystemRegisterTrap,  // Trapped MRS or MSR system register access
                       Exception_ERetTrap,            // Trapped invalid ERET use
                       Exception_InstructionAbort,    // Instruction Abort or Prefetch Abort
                       Exception_PCAlignment,         // PC alignment fault
                       Exception_DataAbort,           // Data Abort
                       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_SoftwareBreakpoint,  // Software Breakpoint Instruction
                       Exception_VectorCatch,         // AArch32 Vector Catch
                       Exception_IRQ,                 // IRQ interrupt
                       Exception_FIQ};                // FIQ interrupt
```

Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```
type ExceptionRecord is (Exception type,            // Exception class
                         bits(25) syndrome,         // Syndrome record
                         bits(64) vaddress,         // Virtual fault address
                         boolean ipavalid,          // Physical fault address is valid
                         bits(52) ipaddress)        // Physical fault address for second stage faults
```

Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception type)

    ExceptionRecord r;

    r.type = type;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.ipaddress = Zeros();

    return r;
```

Library pseudocode for shared/exceptions/traps/ReservedValue

```
// ReservedValue()
// ===============

ReservedValue()

    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();
```

Library pseudocode for shared/exceptions/traps/SystemAccessType

```
enumeration SystemAccessType { SystemAccessType_RT,  SystemAccessType_RRT,  SystemAccessType_DT };
```

Library pseudocode for shared/exceptions/traps/UnallocatedEncoding

```
// UnallocatedEncoding()
// =====================

UnallocatedEncoding()
    if UsingAArch32() && AArch32.ExecutingCP10or11Instr() then
        FPEXC.DEX = '0';
    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();
```

Library pseudocode for shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault type, integer level)

    bits(6) result;
    case type of
        when Fault_AddressSize         result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag          result = '0010':level<1:0>; assert level IN {1,2,3};
        when Fault_Permission          result = '0011':level<1:0>; assert level IN {1,2,3};
        when Fault_Translation         result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal        result = '010000';
        when Fault_SyncExternalOnWalk  result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity          result = '011000';
        when Fault_SyncParityOnWalk    result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity         result = '011001';
        when Fault_AsyncExternal       result = '010001';
        when Fault_Alignment           result = '100001';
        when Fault_Debug               result = '100010';
        when Fault_TLBConflict         result = '110000';
        when Fault_Lockdown            result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive           result = '110101';  // IMPLEMENTATION DEFINED
        otherwise                      Unreachable();

    return result;
```

Library pseudocode for shared/functions/aborts/IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.type != Fault_None;

    if fault.s2fs1walk then
        return fault.type IN {Fault_AccessFlag, Fault_Permission, Fault_Translation,
                              Fault_AddressSize};
    elsif fault.secondstage then
        return fault.type IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
    else
        return FALSE;
```

Library pseudocode for shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.type);
```

Library pseudocode for shared/functions/aborts/IsDebugException

```
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.type != Fault_None;
    return fault.type == Fault_Debug;
```

Library pseudocode for shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =================
// Returns TRUE if the abort currently being processed is an external abort and FALSE otherwise.

boolean IsExternalAbort(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOn
                     Fault_AsyncExternal, Fault_AsyncParity });

// IsExternalAbort()
// =================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.type);
```

Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE other

boolean IsExternalSyncAbort(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOn

// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.type);
```

Library pseudocode for shared/functions/aborts/IsFault

```
// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.type != Fault_None;
```

Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.type);
```

Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.type != Fault_None;

    return fault.secondstage;
```

Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

```
bits(11) LSInstructionSyndrome();
```

Library pseudocode for shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/ASR_C

```
// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    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 - 1 - HighestSetBit(x);
```

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/NOT

```
bits(N) NOT(bits(N) x);
```

Library pseudocode for shared/functions/common/Ones

```
// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// ======

bits(N) Ones()
    return Ones(N);
```

Library pseudocode for shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/ROR_C

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/Replicate

```
// Replicate()
// ===========

bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);
```

Library pseudocode for shared/functions/common/RoundDown

```
integer RoundDown(real x);
```

Library pseudocode for shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

Library pseudocode for shared/functions/common/RoundUp

```
integer RoundUp(real x);
```

Library pseudocode for shared/functions/common/SInt

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

Library pseudocode for shared/functions/common/SignExtend

```
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;

// SignExtend()
// ============

bits(N) SignExtend(bits(M) x)
    return SignExtend(x, N);
```

Library pseudocode for shared/functions/common/UInt

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

Library pseudocode for shared/functions/common/ZeroExtend

```
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;

// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =======

bits(N) Zeros(integer N)
    return Replicate('0',N);

// Zeros()
// =======

bits(N) Zeros()
    return Zeros(N);
```

Library pseudocode for shared/functions/crc/BitReverse

```
// BitReverse()
// ============

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<N-i-1> = data<i>;
    return result;
```

Library pseudocode for shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// ============

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

Library pseudocode for shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
    assert N > 32;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR poly:Zeros(i-32);
    return data<31:0>;
```

Library pseudocode for shared/functions/crypto/AESInvMixColumns

```
bits(128) AESInvMixColumns(bits (128) op);
```

Library pseudocode for shared/functions/crypto/AESInvShiftRows

```
bits(128) AESInvShiftRows(bits(128) op);
```

Library pseudocode for shared/functions/crypto/AESInvSubBytes

```
bits(128) AESInvSubBytes(bits(128) op);
```

Library pseudocode for shared/functions/crypto/AESMixColumns

```
bits(128) AESMixColumns(bits (128) op);
```

Library pseudocode for shared/functions/crypto/AESShiftRows

```
bits(128) AESShiftRows(bits(128) op);
```

Library pseudocode for shared/functions/crypto/AESSubBytes

```
bits(128) AESSubBytes(bits(128) op);
```

Library pseudocode for shared/functions/crypto/HaveChCryptoExt

```
// HaveChCryptoExt()
// =================
// TRUE if SM3 and SM4 extended cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveChCryptoExt()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM3 and SM4 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveCryptoExt

```
// HaveCryptoExt()
// ===============
// TRUE if AES, SHA1, SHA2 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveCryptoExt()
    return boolean IMPLEMENTATION_DEFINED "Has basic Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveCryptoExt2

```
// HaveCryptoExt2()
// ================
// TRUE if SHA512 and SHA3 extended cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveCryptoExt2()
    if !HasArchVersion(ARMv8p2) || !HaveCryptoExt() then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 and SHA3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

Library pseudocode for shared/functions/crypto/SHA256hash

```
// SHA256hash()
// ============

bits(128) SHA256hash(bits (128) X, bits(128) Y, bits(128) W, boolean part1)
    bits(32) chs, maj, t;

    for e = 0 to 3
        chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
        maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
        t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
        X<127:96> = t + X<127:96>;
        Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
        <Y, X> = ROL(Y : X, 32);
    return (if part1 then X else Y);
```

Library pseudocode for shared/functions/crypto/SHAchoose

```
// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// ===============

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// ===============

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

Library pseudocode for shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =============

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));
```

Library pseudocode for shared/functions/crypto/SHAparity

```
// SHAparity()
// ===========

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

Library pseudocode for shared/functions/crypto/Sbox

```
// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = 0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f49

    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// ========================

// Clear the local Exclusives monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

```
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();
```

Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

```
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

```
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// ====================

boolean AArch32.HaveHPDExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// ====================

boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// ================

boolean Have52BitPAExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// ================

boolean Have52BitVAExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// ===============

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// ==============================

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()
// =============
// Returns TRUE if has Dot Product feature support, and FALSE otherwise.

boolean HaveDOTPExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has Dot Product extension";
```

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/HaveFJCVTZSExt

```
// HaveFJCVTZSExt()
// ================

boolean HaveFJCVTZSExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// =============

boolean HaveHPMDExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveNVExt

```
// HaveNVExt()
// ===========

boolean HaveNVExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HavePANExt

```
// HavePANExt()
// ============

boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()
// =================================

boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HavePrivATExt

```
// HavePrivATExt()
// ===============

boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt()
// ================

boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveRASExt

```
// HaveRASExt()
// ============

boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) ||
            boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfiling

```
// HaveStatisticalProfiling()
// ==========================

boolean HaveStatisticalProfiling()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```
// HaveTrapLoadStoreMultipleDeviceExt()
// ====================================

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveUAOExt

```
// HaveUAOExt()
// ============

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt()
// =================

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/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};
    return '0' : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

Library pseudocode for shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = '0011';
        if type1==FPType_SNaN || type2==FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';
    return result;
```

Library pseudocode for shared/functions/float/fpcompareeq/FPCompareEQ

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        if type1==FPType_SNaN || type2==FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;
```

Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

```
// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
    return result;
```

Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
    return result;
```

Library pseudocode for shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding)
    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (type,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if type == FPType_SNaN || type == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        if type == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif type == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCV(value, fpcr, rounding);
    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType 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/FPCRType

```
type FPCRType;
```

Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
    case rm of
        when '00' return FPRounding_TIEAWAY; // A
        when '01' return FPRounding_TIEEVEN; // N
        when '10' return FPRounding_POSINF;  // P
        when '11' return FPRounding_NEGINF;  // M
```

Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// ==================

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF;  // P
        when '10' return FPRounding_NEGINF;  // M
        when '11' return FPRounding_ZERO;    // Z
```

Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN()
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = '0';
    exp  = Ones(E);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpexc/FPExc

```
enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                         FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Ones(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 > value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType_Infinity then
            result = FPInfinity(sign);
        elsif type == FPType_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign);
        else
            result = FPRound(value, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // treat a single quiet-NaN as -Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1');

    return FPMax(op1, op2, fpcr);
```

Library pseudocode for shared/functions/float/fpmin/FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 < value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType_Infinity then
            result = FPInfinity(sign);
        elsif type == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            result = FPRound(value, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // Treat a single quiet-NaN as +Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('0');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('0');

    return FPMin(op1, op2, fpcr);
```

Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd

```
// FPMulAdd()
// ==========
//
// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);

    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity);  zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH

```
// FPMulAddH()
// ===========

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);
    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;
        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');
        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);
        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3, bits(N) op1, bits(N DIV 2)
    assert N IN {32,64};
    bits(N) result;
    if type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
    else
        done = FALSE; result = Zeros(); // 'Don't care' result
    return (done, result);
```

Library pseudocode for shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)
    assert N IN {16,32,64};
    return NOT(op<N-1>) : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)
    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp      cumul = 0;
        when FPExc_DivideByZero   cumul = 1;
        when FPExc_Overflow       cumul = 2;
        when FPExc_Underflow      cumul = 3;
        when FPExc_Inexact        cumul = 4;
        when FPExc_InputDenorm    cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrapException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType type, bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    assert type IN {FPType_QNaN, FPType_SNaN};

    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if type == FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN();
    return result;
```

Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                 bits(N) op1, bits(N) op2,
                                 FPCRType fpcr)
    assert N IN {16,32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

Library pseudocode for shared/functions/float/fpprocessnans3/FPProcessNaNs3

```
// FPProcessNaNs3()
// ================
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRType fpcr)
    assert N IN {16,32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

Library pseudocode for shared/functions/float/fprecipestimate/FPRecipEstimate

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPType_Infinity then
        result = FPZero(sign);
    elsif type == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (
            (N == 16 && Abs(value) < 2.0^-16) ||
            (N == 32 && Abs(value) < 2.0^-128) ||
            (N == 64 && Abs(value) < 2.0^-1024)
          ) then
        case FPRoundingMode(fpcr) of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
          && (
                (N == 16 && Abs(value) >= 2.0^14) ||
                (N == 32 && Abs(value) >= 2.0^126) ||
                (N == 64 && Abs(value) >= 2.0^1022)
             ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign);
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            if fraction<51> == 0 then
                exp = -1;
                fraction = fraction<49:0>:'00';
            else
                fraction = fraction<50:0>:'0';

        integer scaled = UInt('1':fraction<51:44>);

        case N of
            when 16 result_exp =   29 - exp; // In range 29-30 = -1 to 29+1 = 30
            when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
```

```
            when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

        // scaled is in range 256..511 representing a fixed-point number in range [0.5..1.0)
        estimate = RecipEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
        // Convert to scaled floating point result with copied sign bit,
        // high-order bits from estimate, and exponent calculated above.

        fraction = estimate<7:0> : Zeros(44);
        if result_exp == 0 then
            fraction = '1' : fraction<51:1>;
        elsif result_exp == -1 then
            fraction = '01' : fraction<51:2>;
            result_exp = 0;

        case N of
            when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
            when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
            when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

    return result;
```

Library pseudocode for shared/functions/float/fprecipestimate/RecipEstimate

```
// Compute estimate of reciprocal of 9-bit fixed-point number
//
// a is in range 256 .. 511 representing a number in the range 0.5 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipEstimate(integer a)
    assert 256 <= a && a < 512;
    a = a*2+1; // round to nearest
    integer b = (2 ^ 19) DIV a;
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

Library pseudocode for shared/functions/float/fprecpx/FPRecpX

```
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};

    case N of
        when 16 esize =  5;
        when 32 esize =  8;
        when 64 esize = 11;

    bits(N)          result;
    bits(esize)      exp;
    bits(esize)      max_exp;
    bits(N-esize-1)  frac = Zeros();

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp = Ones(esize) - 1;

    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    else
        if IsZero(exp) then // Zero and denormals
            result = sign:max_exp:frac;
        else // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;
```

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, FPCRType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    return FPRoundBase(op, fpcr, rounding);

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Deal with flush-to-zero.
    if ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) && exponent < minimum_exp then
        // Flush-to-zero never generates a trapped exception
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            FPSR.UFC = '1';
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped.
    if biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
        FPProcessException(FPExc_Underflow, fpcr);

    // Round result according to rounding mode.
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up = (error != 0.0 && sign == '1');
```

```
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then      // Rounded up from denormalized to normalized
            biased_exp = 1;
        if int_mant == 2^(F+1) then  // Rounded up to next exponent
            biased_exp = biased_exp + 1;  int_mant = int_mant DIV 2;

    // Handle rounding to odd aka Von Neumann rounding
    if error != 0.0 && rounding == FPRounding_ODD then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
            FPProcessException(FPExc_Overflow, fpcr);
            error = 1.0;  // Ensure that an Inexact exception occurs
        else
            result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;
    else                                     // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0;  // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;

    // Deal with Inexact exception.
    if error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
```

Library pseudocode for shared/functions/float/fpround/FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr, FPRounding rounding)
    fpcr.FZ16 = '0';
    return FPRoundBase(op, fpcr, rounding);
```

Library pseudocode for shared/functions/float/fprounding/FPRounding

```
enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
                         FPRounding_NEGINF,  FPRounding_ZERO,
                         FPRounding_TIEAWAY, FPRounding_ODD};
```

Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

```
// FPRoundingMode()
// ================

// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
    return FPDecodeRounding(fpcr.RMode);
```

Library pseudocode for shared/functions/float/fproundint/FPRoundInt

```
// FPRoundInt()
// ============

// Round OP to nearest integral floating point value using rounding mode ROUNDING.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to OP.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)
    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPType_Infinity then
        result = FPInfinity(sign);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    else
        // extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact
        if real_result == 0.0 then
            result = FPZero(sign);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO);

        // Generate inexact exceptions
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif type == FPType_Infinity then
        result = FPZero('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == 0 do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('01':fraction<51:45>);

        case N of
            when 16 result_exp = (  44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros( 2);
            when 32 result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
            when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);
    return result;
```

Library pseudocode for shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

```
// Compute estimate of reciprocal square root of 9-bit fixed-point number
//
// a is in range 128 .. 511 representing a number in the range 0.25 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipSqrtEstimate(integer a)
    assert 128 <= a && a < 512;
    if a < 256 then // 0.25 .. 0.5
        a = a*2+1;     // a in units of 1/512 rounded to nearest
    else // 0.5 .. 1.0
        a = (a >> 1) << 1;   // discard bottom bit
        a = (a+1)*2;  // a in units of 1/256 rounded to nearest
    integer b = 512;
    while a*(b+1)*(b+1) < 2^28 do
        b = b+1;
    // b = largest b such that b < 2^14 / sqrt(a) do
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    elsif type == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpsub/FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

Library pseudocode for shared/functions/float/fpthree/FPThree

```
// FPThree()
// =========

bits(N) FPThree(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    // If NaN, set cumulative flag or take exception
    if type == FPType_SNaN || type == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

```
// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

bits(N) FPToFixedJS(bits(M) op, FPCRType fpcr, boolean Is64)

    assert M == 64 && N == 32;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    Z = '1';
    // If NaN, set cumulative flag or take exception
    if type == FPType_SNaN || type == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        Z = '0';
    if sign == '1'&& value == 0.0 then
        Z = '0';

    if type == FPType_Infinity then result = 0;

    if Is64 then
        PSTATE.<N,Z,C,V> = '0':Z:'00';
    else
        FPSCR<31:28> = '0':Z:'00';

    return result<31:0>;
```

Library pseudocode for shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =======

bits(N) FPTwo(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fptype/FPType

```
enumeration FPType      {FPType_Nonzero, FPType_Zero, FPType_Infinity,
                         FPType_QNaN, FPType_SNaN};
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// ==========
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpackBase

```
// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr)
    assert N IN {16,32,64};

    if N == 16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            // Produce zero if value is zero or flush-to-zero is selected
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                type = FPType_Zero;  value = 0.0;
            else
                type = FPType_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 then

        sign   = fpval<31>;
        exp32  = fpval<30:23>;
        frac32 = fpval<22:0>;
        if IsZero(exp32) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac32) || fpcr.FZ == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac32) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpcr);
            else
                type = FPType_Nonzero;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64

        sign   = fpval<63>;
        exp64  = fpval<62:52>;
        frac64 = fpval<51:0>;
        if IsZero(exp64) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac64) || fpcr.FZ == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac64) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpcr);
            else
```

```
                type = FPType_Nonzero;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;
    return (type, sign, value);
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr)
    fpcr.FZ16 = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

Library pseudocode for shared/functions/float/fpzero/FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Zeros(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/vfpexpandimm/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);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

Library pseudocode for shared/functions/memory/AArch64.BranchAddr

```
// AArch64.BranchAddr()
// ====================
// Return the virtual address with tag bits removed for storing to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, PSTATE.EL);
    if msbit == 63 then
        return vaddress;
    elsif (PSTATE.EL IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>);
    else
        return ZeroExtend(vaddress<msbit:0>);
```

Library pseudocode for shared/functions/memory/AccType

```
enumeration AccType {AccType_NORMAL, AccType_VEC,         // Normal loads and stores
                     AccType_STREAM, AccType_VECSTREAM,   // Streaming loads and stores
                     AccType_ATOMIC, AccType_ATOMICRW,    // Atomic loads and stores
                     AccType_ORDERED, AccType_ORDEREDRW,  // Load-Acquire and Store-Release
                     AccType_LIMITEDORDERED,              // Load-LOAcquire and Store-LORelease
                     AccType_UNPRIV,                      // Load and store unprivileged
                     AccType_IFETCH,                      // Instruction fetch
                     AccType_PTW,                         // Page table walk
                     // Other operations
                     AccType_DC,                          // Data cache maintenance
                     AccType_IC,                          // Instruction cache maintenance
                     AccType_DCZVA,                       // DC ZVA instructions
                     AccType_AT};                         // Address translation
```

Library pseudocode for shared/functions/memory/AccessDescriptor

```
type AccessDescriptor is (
    AccType acctype,
    boolean page_table_walk,
    boolean secondstage,
    boolean s2fs1walk,
    integer level
)
```

Library pseudocode for shared/functions/memory/AddrTop

```
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        // AArch64 translation regime.
        case regime of
            when EL1
                tbi = (if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0);
                if HavePACExt() then
                    tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
            when EL2
                if HaveVirtHostExt() && ELIsInHost(el) then
                    tbi = (if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0);
                    if HavePACExt() then
                        tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
                else
                    tbi = TCR_EL2.TBI;
                    if HavePACExt() then tbid = TCR_EL2.TBID;
            when EL3
                tbi = TCR_EL3.TBI;
                if HavePACExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr ) then 55 else 63);
```

Library pseudocode for shared/functions/memory/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord      fault,      // fault.type indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    bits(64)         vaddress
)
```

Library pseudocode for shared/functions/memory/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()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

Library pseudocode for shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// ==================

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```

Library pseudocode for shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```

Library pseudocode for shared/functions/memory/CreateAccessDescriptor

```
// CreateAccessDescriptor()
// ========================

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.page_table_walk = FALSE;
    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccessDescriptorPTW

```
// CreateAccessDescriptorPTW()
// ===========================

AccessDescriptor CreateAccessDescriptorPTW(AccType acctype, boolean secondstage,
                                           boolean s2fs1walk, integer level)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.page_table_walk = TRUE;
    accdesc.secondstage = s2fs1walk;
    accdesc.secondstage = secondstage;
    accdesc.level = level;
    return accdesc;
```

Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/memory/DescriptorUpdate

```
type DescriptorUpdate is (
    boolean AF,                 // AF needs to be set
    boolean AP,                 // AP[2] / S2AP[2] will be modified
    AddressDescriptor descaddr  // Descriptor to be updated
)
```

Library pseudocode for shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

Library pseudocode for shared/functions/memory/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_Lockdown,
                   Fault_Exclusive,
                   Fault_ICacheMaint};
```

Library pseudocode for shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault    type,          // Fault Status
                     AccType  acctype,       // Type of access that faulted
                     bits(52) ipaddress,     // Intermediate physical address
                     boolean  s2fs1walk,     // Is on a Stage 1 page table walk
                     boolean  write,         // TRUE for a write, FALSE for a read
                     integer  level,         // For translation, access flag and permission faults
                     bit      extflag,       // IMPLEMENTATION DEFINED syndrome for external aborts
                     boolean  secondstage,   // Is a Stage 2 abort
                     bits(4)  domain,        // Domain number, AArch32 only
                     bits(2)  errortype,      // [ARMv8.2 RAS] AArch32 AET or AArch64 SET
                     bits(4)  debugmoe)      // Debug method of entry, from AArch32 only
```

Library pseudocode for shared/functions/memory/FullAddress

```
type FullAddress is (
    bits(52) physicaladdress,
    bit      NS                  // '0' = Secure, '1' = Non-secure
)
```

Library pseudocode for shared/functions/memory/Hint_Prefetch

```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visiblestate should be on caches and TLBs associated with address, which must be
// accessable by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

Library pseudocode for shared/functions/memory/MBReqDomain

```
enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

Library pseudocode for shared/functions/memory/MBReqTypes

```
enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

Library pseudocode for shared/functions/memory/MemAttrHints

```
type MemAttrHints is (
    bits(2) attrs,  // The possible encodings for each attributes field are as below
    bits(2) hints,  // The possible encodings for the hints are below
    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      type,

    DeviceType   device,     // For Device memory types
    MemAttrHints inner,      // Inner hints and attributes
    MemAttrHints outer,      // Outer hints and attributes

    boolean      shareable,
    boolean      outershareable
)
```

Library pseudocode for shared/functions/memory/Permissions

```
type Permissions is (
    bits(3) ap,   // Access permission bits
    bit     xn,   // Execute-never bit
    bit     xxn,  // [ARMv8.2] Extended execute-never bit for stage 2
    bit     pxn   // Privileged execute-never bit
)
```

Library pseudocode for shared/functions/memory/PrefetchHint

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

Library pseudocode for shared/functions/memory/TLBRecord

```
type TLBRecord is (
    Permissions       perms,
    bit               nG,             // '0' = Global, '1' = not Global
    bits(4)           domain,         // AArch32 only
    boolean           contiguous,     // Contiguous bit from page table
    integer           level,          // AArch32 Short-descriptort format: Indicates Section/Page
    integer           blocksize,      // Describes size of memory translated in KBytes
    DescriptorUpdate  descupdate,     // [ARMv8.1] Context for h/w update of table descriptor
    bit               CnP,            // [ARMv8.2] TLB entry can be shared between different PEs
    AddressDescriptor addrdesc
)
```

Library pseudocode for shared/functions/memory/_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/registers/BranchTo

```
// BranchTo()
// ==========

// Set program counter to a new address, which might include a tag in the top eight bits,
// with a branch reason hint for possible use by hardware fetching the next instruction.

BranchTo(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = AArch64.BranchAddr(target<63:0>);
    return;
```

Library pseudocode for shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// ==============

// Set program counter to a new address, which does not include a tag in the top eight bits,
// with a branch reason hint for possible use by hardware fetching the next instruction.

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_CALL, BranchType_ERET, BranchType_DBGEXIT,
                        BranchType_RET, BranchType_JMP, BranchType_EXCEPTION,
                        BranchType_UNKNOWN};
```

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 next instruction.
bits(N) NextInstrAddr();
```

Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

```
// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);
```

Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr()
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

Library pseudocode for shared/functions/registers/_PC

```
bits(64) _PC;
```

Library pseudocode for shared/functions/registers/_R

```
array bits(64) _R[0..30];
```

Library pseudocode for shared/functions/registers/_V

```
array bits(128) _V[0..31];
```

Library pseudocode for shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// ============================

bits(32) SPSR[]
    bits(32) result;
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ       result = SPSR_fiq;
            when M32_IRQ       result = SPSR_irq;
            when M32_Svc       result = SPSR_svc;
            when M32_Monitor   result = SPSR_mon;
            when M32_Abort     result = SPSR_abt;
            when M32_Hyp       result = SPSR_hyp;
            when M32_Undef     result = SPSR_und;
            otherwise          Unreachable();
    else
        case PSTATE.EL of
            when EL1           result = SPSR_EL1;
            when EL2           result = SPSR_EL2;
            when EL3           result = SPSR_EL3;
            otherwise          Unreachable();

    return result;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(32) value
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ       SPSR_fiq = value;
            when M32_IRQ       SPSR_irq = value;
            when M32_Svc       SPSR_svc = value;
            when M32_Monitor   SPSR_mon = value;
            when M32_Abort     SPSR_abt = value;
            when M32_Hyp       SPSR_hyp = value;
            when M32_Undef     SPSR_und = value;
            otherwise          Unreachable();
    else
        case PSTATE.EL of
            when EL1           SPSR_EL1 = value;
            when EL2           SPSR_EL2 = value;
            when EL3           SPSR_EL3 = value;
            otherwise          Unreachable();

    return;
```

Library pseudocode for shared/functions/system/ArchVersion

```
enumeration ArchVersion {
    ARMv8p0
    , ARMv8p1
    , ARMv8p2
    , ARMv8p3
};
```

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/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/CurrentInstrSet

```
// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()

    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;
```

Library pseudocode for shared/functions/system/CurrentPL

```
// CurrentPL()
// ===========

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);
```

Library pseudocode for shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

Library pseudocode for shared/functions/system/ELFromM32

```
// ELFromM32()
// ===========

(boolean,bits(2)) ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid,EL):
    //   'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    //           and the current value of SCR.NS/SCR_EL3.NS.
    //   'EL'    is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = !BadMode(mode);  // Check for modes that are not valid for this implementation
    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
            valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && HighestELUsingAArch32() && SCR.NS == '0' then EL3 else EL1);
        when M32_User
            el = EL0;
        otherwise
            valid = FALSE;           // Passed an illegal mode value
    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

Library pseudocode for shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(32) spsr)
    if spsr<4> == '0' then                              // AArch64 state
        el = spsr<3:2>;
        if HighestELUsingAArch32() then                 // No AArch64 support
            valid = FALSE;
        elsif !HaveEL(el) then                          // Exception level not implemented
            valid = FALSE;
        elsif spsr<1> == '1' then                       // M[1] must be 0
            valid = FALSE;
        elsif el == EL0 && spsr<0> == '1' then          // for EL0, M[0] must be 0
            valid = FALSE;
        elsif el == EL2 && HaveEL(EL3) && SCR_EL3.NS == '0' then
            valid = FALSE;                              // EL2 only valid in Non-secure state
        else
            valid = TRUE;
    elsif !HaveAnyAArch32() then                        // AArch32 not supported
        valid = FALSE;
    else                                                // AArch32 state
        (valid, el) = ELFromM32(spsr<4:0>);
    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
```

Library pseudocode for shared/functions/system/ELIsInHost

```
// ELIsInHost()
// ============
// Returns TRUE if HaveVirtHostExt() is TRUE and executing at 'el' would mean be executing within
// an EL2 Host OS or an EL0 application of a Host OS in Non-secure state using AArch64 with
// HCR_EL2.E2H set to 1, and FALSE otherwise.

boolean ELIsInHost(bits(2) el)
    return (!IsSecureBelowEL3() && HaveVirtHostExt() && !ELUsingAArch32(EL2) &&
            HCR_EL2.E2H == '1' && (el == EL2 || (el == EL0 && HCR_EL2.TGE == '1')));
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====================

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// ======================

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    // Returns (known, aarch32):
    //   'known'   is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
    //             using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
    //   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
    boolean aarch32;
    known = TRUE;
    if !HaveAArch32EL(el) then
        aarch32 = FALSE;                          // Exception level is using AArch64
    elsif HighestELUsingAArch32() then
        aarch32 = TRUE;                           // All levels are using AArch32
    else
        aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0';

        aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) && !secure && HCR_EL2.RW == '0' &&
                          !(HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt())));
        if el == EL0 && !aarch32_at_el1 then      // Only know if EL0 using AArch32 from PSTATE
            if PSTATE.EL == EL0 then
                aarch32 = PSTATE.nRW == '1';      // EL0 controlled by PSTATE
            else
                known = FALSE;                    // EL0 state is UNKNOWN
        else
            aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});
    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);
```

Library pseudocode for shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/EndOfInstruction

```
// Terminate processing of the current instruction.
EndOfInstruction();
```

Library pseudocode for shared/functions/system/EnterLowPowerState

```
// PE enters a low-power state
EnterLowPowerState();
```

Library pseudocode for shared/functions/system/ErrorSynchronizationBarrier

```
// Perform an Error Synchronization Barrier operation
ErrorSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(32) GetPSRFromPSTATE()
    bits(32) spsr = Zeros();
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<21>    = PSTATE.SS;
    spsr<20>    = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        spsr<27>    = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if HaveUAOExt() then spsr<23> = '0';
        spsr<19:16> = PSTATE.GE;
        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9>     = PSTATE.E;
        spsr<8:6>   = PSTATE.<A,I,F>;                   // No PSTATE.D in AArch32 state
        spsr<5>     = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW;               // bit [4] is the discriminator
        spsr<4:0>   = PSTATE.M;
    else // AArch64 state
        if HaveUAOExt() then spsr<23> = PSTATE.UAO;
        spsr<9:6>   = PSTATE.<D,A,I,F>;
        spsr<4>     = PSTATE.nRW;
        spsr<3:2>   = PSTATE.EL;
        spsr<0>     = PSTATE.SP;
    return spsr;
```

Library pseudocode for shared/functions/system/HasArchVersion

```
// HasArchVersion()
// ================

// Return TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// ===============

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE;                    // The Exception level is not implemented
    elsif !HaveAnyAArch32() then
        return FALSE;                    // No Exception level can use AArch32
    elsif HighestELUsingAArch32() then
        return TRUE;                     // All Exception levels are using AArch32
    elsif el == HighestEL() then
        return FALSE;                    // The highest Exception level is using AArch64
    elsif el == EL0 then
        return TRUE;                     // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAnyAArch32

```
// HaveAnyAArch32()
// ================
// Return TRUE if AArch32 state is supported at any Exception level

boolean HaveAnyAArch32()
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAnyAArch64

```
// HaveAnyAArch64()
// ================
// Return TRUE if AArch64 state is supported at any Exception level

boolean HaveAnyAArch64()
    return !HighestELUsingAArch32();
```

Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    if el IN {EL1,EL0} then
        return TRUE;                             // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =============
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HighestEL

```
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

Library pseudocode for shared/functions/system/HighestELUsingAArch32

```
// HighestELUsingAArch32()
// =======================
// Return TRUE if configured to boot into AArch32 operation

boolean HighestELUsingAArch32()
    if !HaveAnyAArch32() then return FALSE;
    return boolean IMPLEMENTATION_DEFINED;       // e.g. CFG32SIGNAL == HIGH
```

Library pseudocode for shared/functions/system/Hint_Yield

```
Hint_Yield();
```

Library pseudocode for shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(32) spsr)

    // Check for return:
    //   * To an unimplemented Exeception level.
    //   * To EL2 in Secure state.
    //   * 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 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 return to EL1 in Non-secure state when HCR.TGE is set
    if HaveEL(EL2) && target == EL1 && !IsSecureBelowEL3() && HCR_EL2.TGE == '1' 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

```
// Return TRUE if there are any pending physical or virtual interrupts, and FALSE otherwise
boolean InterruptPending();
```

Library pseudocode for shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE otherwise

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

Library pseudocode for shared/functions/system/IsInHost

```
// IsInHost()
// ==========
// Returns TRUE if HaveVirtHostExt() is TRUE and executing within a Host OS or an EL0 application
// of a Host OS using AArch64 with HCR_EL2.E2H set to 1, and FALSE otherwise.

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

Library pseudocode for shared/functions/system/IsSecure

```
// IsSecure()
// ==========

boolean IsSecure()
    // Return TRUE if current Exception level is in Secure state.
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elsif 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';
    elsif HaveEL(EL2) then
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure;
        return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/Mode_Bits

```
constant bits(5) M32_User    = '10000';
constant bits(5) M32_FIQ     = '10001';
constant bits(5) M32_IRQ     = '10010';
constant bits(5) M32_Svc     = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort   = '10111';
constant bits(5) M32_Hyp     = '11010';
constant bits(5) M32_Undef   = '11011';
constant bits(5) M32_System  = '11111';
```

Library pseudocode for shared/functions/system/PLOfEL

```
// PLOfEL()
// ========

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3  return if HighestELUsingAArch32() then PL1 else PL3;
        when EL2  return PL2;
        when EL1  return PL1;
        when EL0  return PL0;
```

Library pseudocode for shared/functions/system/PSTATE

```
ProcState PSTATE;
```

Library pseudocode for shared/functions/system/PrivilegeLevel

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

Library pseudocode for shared/functions/system/ProcState

```
type ProcState is (
    bits (1) N,        // Negative condition flag
    bits (1) Z,        // Zero condition flag
    bits (1) C,        // Carry condition flag
    bits (1) V,        // oVerflow condition flag
    bits (1) D,        // Debug mask bit                     [AArch64 only]
    bits (1) A,        // SError interrupt mask bit
    bits (1) I,        // IRQ mask bit
    bits (1) F,        // FIQ mask bit
    bits (1) PAN,      // Privileged Access Never Bit        [v8.1]
    bits (1) UAO,      // User Access Override               [v8.2]
    bits (1) 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 (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(32) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

Library pseudocode for shared/functions/system/SCRType

```
type SCRType;
```

Library pseudocode for shared/functions/system/SCR_GEN

```
// SCR_GEN[]
// =========

SCRType SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(32) r;
    if HighestELUsingAArch32() then
        r = SCR;
    else
        r = SCR_EL3;
    return r;
```

Library pseudocode for shared/functions/system/SErrorPending

```
// Return TRUE if a physical SError interrupt is pending; that is, if ISR_EL1.A == 1
boolean SErrorPending();
```

Library pseudocode for shared/functions/system/SendEvent

```
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed
SendEvent();
```

Library pseudocode for shared/functions/system/SendEventLocal

```
// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed

SendEventLocal()
    EventRegister = '1';
    return;
```

Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(32) spsr)

    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then                    // AArch32 state
            AArch32.WriteMode(spsr<4:0>);         // Sets PSTATE.EL correctly
        else                                      // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL  = spsr<3:2>;
            PSTATE.SP  = spsr<0>;
    // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
    // the T bit is set to zero or copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';

    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if PSTATE.nRW == '1' then                     // AArch32 state
        PSTATE.Q         = spsr<27>;
        PSTATE.IT        = RestoredITBits(spsr);
        PSTATE.GE        = spsr<19:16>;
        PSTATE.E         = spsr<9>;
        PSTATE.<A,I,F>   = spsr<8:6>;             // No PSTATE.D in AArch32 state
        PSTATE.T         = spsr<5>;               // PSTATE.J is RES0
    else                                          // AArch64 state
        PSTATE.<D,A,I,F> = spsr<9:6>;             // No PSTATE.<Q,IT,GE,E,T> in AArch64 state

    if HavePANExt() then PSTATE.<PAN> = spsr<22>;
    if HaveUAOExt() then PSTATE.UAO = spsr<23>;
    return;
```

Library pseudocode for shared/functions/system/SynchronizeContext

```
SynchronizeContext();
```

Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.
TakeUnmaskedSErrorInterrupts();
```

Library pseudocode for shared/functions/system/ThisInstr

```
bits(32) ThisInstr();
```

Library pseudocode for shared/functions/system/ThisInstrLength

```
integer ThisInstrLength();
```

Library pseudocode for shared/functions/system/Unreachable

```
Unreachable()
    assert FALSE;
```

Library pseudocode for shared/functions/system/UsingAArch32

```
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAnyAArch32() then assert !aarch32;
    if HighestELUsingAArch32() then assert aarch32;
    return aarch32;
```

Library pseudocode for shared/functions/system/WaitForEvent

```
// WaitForEvent()
// ==============
// PE suspends its operation and enters a low-power state
// if the Event Register is clear when the WFE is executed

WaitForEvent()
    if EventRegister == '0' then
        EnterLowPowerState();
    return;
```

Library pseudocode for shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()
// ==================
// PE suspends its operation to enter a low-power state
// until a WFI wake-up event occurs or the PE is reset

WaitForInterrupt()
    EnterLowPowerState();
    return;
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictable

```
// ConstrainUnpredictable()
// ========================
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.

Constraint ConstrainUnpredictable(Unpredictable which)
    case which of
        when Unpredictable_WBOVERLAPLD
            return Constraint_WBSUPPRESS; // return loaded value
        when Unpredictable_WBOVERLAPST
            return Constraint_NONE;     // store pre-writeback value
        when Unpredictable_LDPOVERLAP
            return Constraint_UNDEF;    // instruction is UNDEFINED
        when Unpredictable_BASEOVERLAP
            return Constraint_NONE;     // use original address
        when Unpredictable_DATAOVERLAP
            return Constraint_NONE;     // store original value
        when Unpredictable_DEVPAGE2
            return Constraint_FAULT;    // take an alignment fault
        when Unpredictable_INSTRDEVICE
            return Constraint_NONE;     // Do not take a fault
        when Unpredictable_RESCPACR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESMAIR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESTEXCB
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESDACR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESPRRR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESVTCRS
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESTnSZ
            return Constraint_FORCE;    // Map to the limit value
        when Unpredictable_OORTnSZ
            return Constraint_FORCE;    // Map to the limit value
        when Unpredictable_LARGEIPA
            return Constraint_FORCE;    // Restrict the inputsize 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_AFUPDATE     // AF update for alignment or permission fault
            return Constraint_TRUE;
        when Unpredictable_IESBinDebug  // Use SCTLR[].IESB in Debug state
            return Constraint_TRUE;
        when Unpredictable_CLEARERRITEZERO // Clearing sticky errors when instruction in flight
            return Constraint_FALSE;
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

```
// ConstrainUnpredictableBits()
// ============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint_UNKNOWN then
        return (c, Zeros(width));           // See notes; this is an example implementation only
    else
        return (c, bits(width) UNKNOWN);    // bits result not used
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

boolean ConstrainUnpredictableBool(Unpredictable which)

    c = ConstrainUnpredictable(which);
    assert c IN {Constraint_TRUE, Constraint_FALSE};
    return (c == Constraint_TRUE);
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// ===============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the integer part
// of the result.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high, Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint_UNKNOWN then
        return (c, low);                // See notes; this is an example implementation only
    else
        return (c, integer UNKNOWN);    // integer result not used
```

Library pseudocode for shared/functions/unpredictable/Constraint

```
enumeration Constraint    {// General:
                           Constraint_NONE,              // Instruction executes with
                                                         //  no change or side-effect to its described
                           Constraint_UNKNOWN,           // Destination register has UNKNOWN value
                           Constraint_UNDEF,             // Instruction is UNDEFINED
                           Constraint_UNDEFEL0,          // Instruction is UNDEFINED at EL0 only
                           Constraint_NOP,               // Instruction executes as NOP
                           Constraint_TRUE,
                           Constraint_FALSE,
                           Constraint_DISABLED,
                           Constraint_UNCOND,            // Instruction executes unconditionally
                           Constraint_COND,              // Instruction executes conditionally
                           Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
                           // Load-store:
                           Constraint_WBSUPPRESS, Constraint_FAULT,
                           // IPA too large
                           Constraint_FORCE, Constraint_FORCENOSLCHECK};
```

Library pseudocode for shared/functions/unpredictable/Unpredictable

```
enumeration Unpredictable {// Writeback/transfer register overlap (load):
                           Unpredictable_WBOVERLAPLD,
                           // Writeback/transfer register overlap (store):
                           Unpredictable_WBOVERLAPST,
                           // Load Pair transfer register overlap:
                           Unpredictable_LDPOVERLAP,
                           // Store-exclusive base/status register overlap
                           Unpredictable_BASEOVERLAP,
                           // Store-exclusive data/status register overlap
                           Unpredictable_DATAOVERLAP,
                           // Load-store alignment checks:
                           Unpredictable_DEVPAGE2,
                           // Instruction fetch from Device memory
                           Unpredictable_INSTRDEVICE,
                           // Reserved CPACR value
                           Unpredictable_RESCPACR,
                           // Reserved MAIR value
                           Unpredictable_RESMAIR,
                           // Reserved TEX:C:B value
                           Unpredictable_RESTEXCB,
                           // Reserved PRRR value
                           Unpredictable_RESPRRR,
                           // Reserved DACR field
                           Unpredictable_RESDACR,
                           // Reserved VTCR.S value
                           Unpredictable_RESVTCRS,
                           // Reserved TCR.TnSZ value
                           Unpredictable_RESTnSZ,
                           // Out-of-range TCR.TnSZ value
                           Unpredictable_OORTnSZ,
                           // IPA size exceeds PA size
                           Unpredictable_LARGEIPA,
                           // Syndrome for a known-passing conditional A32 instruction
                           Unpredictable_ESRCONDPASS,
                           // Illegal State exception: zero PSTATE.IT
                           Unpredictable_ILZEROIT,
                           // Illegal State exception: zero PSTATE.T
                           Unpredictable_ILZEROT,
                           // Debug: prioritization of Vector Catch
                           Unpredictable_BPVECTORCATCHPRI,
                           // Debug Vector Catch: match on 2nd halfword
                           Unpredictable_VCMATCHHALF,
                           // Debug Vector Catch: match on Data Abort or Prefetch abort
                           Unpredictable_VCMATCHDAPA,
                           // Debug watchpoints: non-zero MASK and non-ones BAS
                           Unpredictable_WPMASKANDBAS,
                           // Debug watchpoints: non-contiguous BAS
                           Unpredictable_WPBASCONTIGUOUS,
                           // Debug watchpoints: reserved MASK
                           Unpredictable_RESWPMASK,
                           // Debug watchpoints: non-zero MASKed bits of address
                           Unpredictable_WPMASKEDBITS,
                           // Debug breakpoints and watchpoints: reserved control bits
                           Unpredictable_RESBPWPCTRL,
                           // Debug breakpoints: not implemented
                           Unpredictable_BPNOTIMPL,
                           // Debug breakpoints: reserved type
                           Unpredictable_RESBPTYPE,
                           // Debug breakpoints: not-context-aware breakpoint
                           Unpredictable_BPNOTCTXCMP,
                           // Debug breakpoints: match on 2nd halfword of instruction
                           Unpredictable_BPMATCHHALF,
                           // Debug breakpoints: mismatch on 2nd halfword of instruction
                           Unpredictable_BPMISMATCHHALF,
                           // Debug: restart to a misaligned AArch32 PC value
                           Unpredictable_RESTARTALIGNPC,
                           // Debug: restart to a not-zero-extended AArch32 PC value
                           Unpredictable_RESTARTZEROUPPERPC,
                           // Zero top 32 bits of X registers in AArch32 state:
                           Unpredictable_ZEROUPPER,
```

```
                                // Zero top 32 bits of PC on illegal return to AArch32 state
                                Unpredictable_ERETZEROUPPERPC,
                                // Force address to be aligned when interworking branch to A32 state
                                Unpredictable_A32FORCEALIGNPC,
                                // SMC disabled
                                Unpredictable_SMD,
                                // Access Flag Update by HW
                                Unpredictable_AFUPDATE,
                                // Consider SCTLR[].IESB in Debug state
                                Unpredictable_IESBinDebug,
                                // No events selected in PMSEVFR_EL1
                                Unpredictable_ZEROPMSEVFR,
                                // No operation type selected in PMSFCR_EL1
                                Unpredictable_NOOPTYPES,
                                // Zero latency in PMSLATFR_EL1
                                Unpredictable_ZEROMINLATENCY,
                                // Clearing DCC/ITR sticky flags when instruction is in flight
                                Unpredictable_CLEARERRITEZERO,
                                // To be determined -- THESE SHOULD BE RESOLVED (catch-all)
                                Unpredictable_TBD};
```

Library pseudocode for shared/functions/vector/AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);

    return imm64;
```

Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

Library pseudocode for shared/functions/vector/SatQ

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```

Library pseudocode for shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N IN {16,32};
    if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)

        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        case N of
            when 16 estimate = RecipSqrtEstimate(UInt(operand<15:7>));
            when 32 estimate = RecipSqrtEstimate(UInt(operand<31:23>));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N IN {16,32};
    if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        case N of
            when 16 estimate = RecipEstimate(UInt(operand<15:7>));
            when 32 estimate = RecipEstimate(UInt(operand<31:23>));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

Library pseudocode for shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

Library pseudocode for shared/translation/attrs/CombineS1S2AttrHints

```
// CombineS1S2AttrHints()
// ======================

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc)

    MemAttrHints result;

    if s2desc.attrs == '01' || s1desc.attrs == '01' then
        result.attrs = bits(2) UNKNOWN;   // Reserved
    elsif s2desc.attrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
        result.attrs = MemAttr_NC;        // Non-cacheable
    elsif s2desc.attrs == MemAttr_WT || s1desc.attrs == MemAttr_WT then
        result.attrs = MemAttr_WT;        // Write-through
    else
        result.attrs = MemAttr_WB;        // Write-back

    result.hints = s1desc.hints;
    result.transient = s1desc.transient;

    return result;
```

Library pseudocode for shared/translation/attrs/CombineS1S2Desc

```
// CombineS1S2Desc()
// =================
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    AddressDescriptor result;

    result.paddress = s2desc.paddress;

    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    elsif s2desc.memattrs.type == MemType_Device || s1desc.memattrs.type == MemType_Device then
        result.memattrs.type = MemType_Device;
        if s1desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s2desc.memattrs.device;
        elsif s2desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s1desc.memattrs.device;
        else                      // Both Device
            result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
                                                       s2desc.memattrs.device);
    else                          // Both Normal
        result.memattrs.type = MemType_Normal;
        result.memattrs.device = DeviceType UNKNOWN;
        result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner);
        result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer);
        result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
        result.memattrs.outershareable = (s1desc.memattrs.outershareable ||
                                          s2desc.memattrs.outershareable);

    result.memattrs = MemAttrDefaults(result.memattrs);

    return result;
```

Library pseudocode for shared/translation/attrs/CombineS1S2Device

```
// CombineS1S2Device()
// ===================
// Combines device types from stage 1 and stage 2

DeviceType CombineS1S2Device(DeviceType s1device, DeviceType s2device)

    if s2device == DeviceType_nGnRnE || s1device == DeviceType_nGnRnE then
        result = DeviceType_nGnRnE;
    elsif s2device == DeviceType_nGnRE || s1device == DeviceType_nGnRE then
        result = DeviceType_nGnRE;
    elsif s2device == DeviceType_nGRE || s1device == DeviceType_nGRE then
        result = DeviceType_nGRE;
    else
        result = DeviceType_GRE;

    return result;
```

Library pseudocode for shared/translation/attrs/LongConvertAttrsHints

```
// LongConvertAttrsHints()
// =======================
// Convert the long attribute fields for Normal memory as used in the MAIR fields
// to orthogonal attributes and hints

MemAttrHints LongConvertAttrsHints(bits(4) attrfield, AccType acctype)
    assert !IsZero(attrfield);
    MemAttrHints result;
    if S1CacheDisabled(acctype) then             // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        if attrfield<3:2> == '00' then          // Write-through transient
            result.attrs = MemAttr_WT;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        elsif attrfield<3:0> == '0100' then     // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
            result.transient = FALSE;
        elsif attrfield<3:2> == '01' then       // Write-back transient
            result.attrs = attrfield<1:0>;
            result.hints = MemAttr_WB;
            result.transient = TRUE;
        else                                    // Write-through/Write-back non-transient
            result.attrs = attrfield<3:2>;
            result.hints = attrfield<1:0>;
            result.transient = FALSE;

    return result;
```

Library pseudocode for shared/translation/attrs/MemAttrDefaults

```
// MemAttrDefaults()
// =================
// Supply default values for memory attributes, including overriding the shareability attributes
// for Device and Non-cacheable memory types.

MemoryAttributes MemAttrDefaults(MemoryAttributes memattrs)

    if memattrs.type == MemType_Device then
        memattrs.inner = MemAttrHints UNKNOWN;
        memattrs.outer = MemAttrHints UNKNOWN;
        memattrs.shareable = TRUE;
        memattrs.outershareable = TRUE;
    else
        memattrs.device = DeviceType UNKNOWN;
        if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
            memattrs.shareable = TRUE;
            memattrs.outershareable = TRUE;

    return memattrs;
```

Library pseudocode for shared/translation/attrs/S1CacheDisabled

```
// S1CacheDisabled()
// =================

boolean S1CacheDisabled(AccType acctype)
    if ELUsingAArch32(S1TranslationRegime()) then
        if PSTATE.EL == EL2 then
            enable = if acctype == AccType_IFETCH then HSCTLR.I else HSCTLR.C;
        else
            enable = if acctype == AccType_IFETCH then SCTLR.I else SCTLR.C;
    else
        enable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;
    return enable == '0';
```

Library pseudocode for shared/translation/attrs/S2AttrDecode

```
// S2AttrDecode()
// ==============
// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)

    MemoryAttributes memattrs;

    if attr<3:2> == '00' then                    // Device
        memattrs.type = MemType_Device;
        case attr<1:0> of
            when '00'  memattrs.device = DeviceType_nGnRnE;
            when '01'  memattrs.device = DeviceType_nGnRE;
            when '10'  memattrs.device = DeviceType_nGRE;
            when '11'  memattrs.device = DeviceType_GRE;

    elsif attr<1:0> != '00' then                // Normal
        memattrs.type = MemType_Normal;
        memattrs.outer = S2ConvertAttrsHints(attr<3:2>, acctype);
        memattrs.inner = S2ConvertAttrsHints(attr<1:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';

    else
        memattrs = MemoryAttributes UNKNOWN;    // Reserved

    return MemAttrDefaults(memattrs);
```

Library pseudocode for shared/translation/attrs/S2CacheDisabled

```
// S2CacheDisabled()
// =================

boolean S2CacheDisabled(AccType acctype)
    if ELUsingAArch32(EL2) then
        disable = if acctype == AccType_IFETCH then HCR2.ID else HCR2.CD;
    else
        disable = if acctype == AccType_IFETCH then HCR_EL2.ID else HCR_EL2.CD;

    return disable == '1';
```

Library pseudocode for shared/translation/attrs/S2ConvertAttrsHints

```
// S2ConvertAttrsHints()
// =====================
// Converts the attribute fields for Normal memory as used in stage 2
// descriptors to orthogonal attributes and hints

MemAttrHints S2ConvertAttrsHints(bits(2) attr, AccType acctype)
    assert !IsZero(attr);

    MemAttrHints result;

    if S2CacheDisabled(acctype) then   // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case attr of
            when '01'                   // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '10'                   // Write-through
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RWA;
            when '11'                   // Write-back
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;

    result.transient = FALSE;

    return result;
```

Library pseudocode for shared/translation/attrs/ShortConvertAttrsHints

```
// ShortConvertAttrsHints()
// ========================
// Converts the short attribute fields for Normal memory as used in the TTBR and
// TEX fields to orthogonal attributes and hints

MemAttrHints ShortConvertAttrsHints(bits(2) RGN, AccType acctype, boolean secondstage)

    MemAttrHints result;

    if (!secondstage && S1CacheDisabled(acctype)) || (secondstage && S2CacheDisabled(acctype)) then
       // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case RGN of
            when '00'                   // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '01'                   // Write-back, Read and Write allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;
            when '10'                   // Write-through, Read allocate
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RA;
            when '11'                   // Write-back, Read allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RA;

    result.transient = FALSE;

    return result;
```

Library pseudocode for shared/translation/attrs/WalkAttrDecode

```
// WalkAttrDecode()
// ================

MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN, boolean secondstage)

    MemoryAttributes mematrs;

    AccType acctype = AccType_NORMAL;

    mematrs.type = MemType_Normal;
    mematrs.inner = ShortConvertAttrsHints(IRGN, acctype, secondstage);
    mematrs.outer = ShortConvertAttrsHints(ORGN, acctype, secondstage);
    mematrs.shareable = SH<1> == '1';
    mematrs.outershareable = SH == '10';

    return MemAttrDefaults(mematrs);
```

Library pseudocode for shared/translation/translation/HasS2Translation

```
// HasS2Translation()
// ==================
// Returns TRUE if stage 2 translation is present for the current translation regime

boolean HasS2Translation()
    return (HaveEL(EL2) && !IsSecure() && !IsInHost() && PSTATE.EL IN {EL0,EL1});
```

Library pseudocode for shared/translation/translation/Have16bitVMID

```
// Have16bitVMID()
// ===============
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return HaveEL(EL2) && boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/translation/translation/IsEL1TransRegimeRegs

```
// IsEL1TransRegimeRegs()
// ======================

// Returns TRUE if its a register in EL1 translation regime

boolean IsEL1TransRegimeRegs()
    return !HaveEL(EL2) || PSTATE.EL == EL1 || (PSTATE.EL == EL0 && (HCR_EL2.E2H == '0' ||  HCR_EL2.TGE
```

Library pseudocode for shared/translation/translation/PAMax

```
// PAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the physical address
// size for this processor, as log2().

integer PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

Library pseudocode for shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

Library pseudocode for shared/translation/translation/VAMax

```
// VAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the virtual address
// size for this processor, as log2().

integer VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.

Library pseudocode for aarch64/functions/pac/auth/Auth

```
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit keynumber)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = CalculateTBI(ptr, data);
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr, ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    if tbi then
        original_ptr = ptr<63:56>:extfield<56-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<64-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
            result = original_ptr;
        else
            error_code = keynumber:NOT(keynumber);
            result = original_ptr<63:55>:error_code:original_ptr<52:0>;
    else
        if ((PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit>) &&
            (PAC<63:56> == ptr<63:56>)) then
            result = original_ptr;
        else
            error_code = keynumber:NOT(keynumber);
            result = original_ptr<63>:error_code:original_ptr<60:0>;
    return result;
```

Library pseudocode for aarch64/functions/pac/authda/AuthDA

```
// AuthDA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APDAKey_EL1, TRUE, '0');
```

Library pseudocode for aarch64/functions/pac/authdb/AuthDB

```
// AuthDB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APDBKey_EL1, TRUE, '1');
```

Library pseudocode for aarch64/functions/pac/authia/AuthIA

```
// AuthIA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APIAKey_EL1, FALSE, '0');
```

Library pseudocode for aarch64/functions/pac/authib/AuthIB

```
// AuthIB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return Auth(X, Y, APIBKey_EL1, FALSE, '1');
```

Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =======================

integer CalculateBottomPACBit(bits(64) ptr, bit top_bit)
    integer tsz_field;

    if PtrHasUpperAndLowerAddRanges() then
        if IsEL1TransRegimeRegs() then
            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 == '11';
        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 == '11';
    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 == '11' else TCR_EL3.TG0 == '11';

    max_limit_tsz_field = 39;
    if tsz_field > max_limit_tsz_field then
        // TCR_ELx.TySZ is out of range
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
    tszmin = if using64k && VAMax() == 52 then 12 else 16;
    if tsz_field < tszmin then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = tszmin;
    return (64-tsz_field);
```

Library pseudocode for aarch64/functions/pac/calculatetbi/CalculateTBI

```
// CalculateTBI()
// ==============

boolean CalculateTBI(bits(64) ptr, boolean data)
    boolean tbi = FALSE;

    if PtrHasUpperAndLowerAddRanges() then
        if IsEL1TransRegimeRegs() then
            if data then
                tbi = if ptr<55> == '1' then TCR_EL1.TBI1 == '1' else TCR_EL1.TBI0 == '1';
            else
                if ptr<55> == '1' then
                    tbi = TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0';
                else
                    tbi = TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0';
        else
            // EL2 translation regime registers
            if data then
                tbi = if ptr<55> == '1' then TCR_EL2.TBI1 == '1' else TCR_EL2.TBI0 == '1';
            else
                if ptr<55> == '1' then
                    tbi = TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0';
                else
                    tbi = TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0';
    elsif PSTATE.EL == EL2 then
        tbi = if data then TCR_EL2.TBI=='1' else TCR_EL2.TBI=='1' && TCR_EL2.TBID=='0';
    elsif PSTATE.EL == EL3 then
        tbi = if data then TCR_EL3.TBI=='1' else TCR_EL3.TBI=='1' && TCR_EL3.TBID=='0';

    return tbi;
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```
array bits(64) RC[0..4];

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
    bits(64)  workingval;
    bits(64)  runningmod;
    bits(64)  roundkey;
    bits(64)  modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;
    RC[3] = 0x082EFA98EC4E6C89<63:0>;
    RC[4] = 0x452821E638D01377<63:0>;

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;
    for i = 0 to 4
        roundkey = key1 EOR runningmod;
        workingval  = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
        workingval = PACSub(workingval);
        runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = PACSub(workingval);
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 EOR workingval;
    workingval = PACCellInvShuffle(workingval);
    workingval = PACInvSub(workingval);
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    workingval = workingval EOR key0;
    workingval = workingval EOR runningmod;
    for i = 0 to 4
        workingval = PACInvSub(workingval);
        if i < 4 then
            workingval = PACMult(workingval);
            workingval = PACCellInvShuffle(workingval);
        runningmod = TweakInvShuffle(runningmod<63:0>);
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR RC[4-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```
// PACCellInvShuffle()
// ===================

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

```
// PACCellShuffle()
// ================

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// ===========

bits(64) PACInvSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher

    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '0101';
            when '0001'  Toutput<4*i+3:4*i> = '1110';
            when '0010'  Toutput<4*i+3:4*i> = '1101';
            when '0011'  Toutput<4*i+3:4*i> = '1000';
            when '0100'  Toutput<4*i+3:4*i> = '1010';
            when '0101'  Toutput<4*i+3:4*i> = '1011';
            when '0110'  Toutput<4*i+3:4*i> = '0001';
            when '0111'  Toutput<4*i+3:4*i> = '1001';
            when '1000'  Toutput<4*i+3:4*i> = '0010';
            when '1001'  Toutput<4*i+3:4*i> = '0110';
            when '1010'  Toutput<4*i+3:4*i> = '1111';
            when '1011'  Toutput<4*i+3:4*i> = '0000';
            when '1100'  Toutput<4*i+3:4*i> = '0100';
            when '1101'  Toutput<4*i+3:4*i> = '1100';
            when '1110'  Toutput<4*i+3:4*i> = '0111';
            when '1111'  Toutput<4*i+3:4*i> = '0011';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)
    bits(4)  t0;
    bits(4)  t1;
    bits(4)  t2;
    bits(4)  t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '1011';
            when '0001'  Toutput<4*i+3:4*i> = '0110';
            when '0010'  Toutput<4*i+3:4*i> = '1000';
            when '0011'  Toutput<4*i+3:4*i> = '1111';
            when '0100'  Toutput<4*i+3:4*i> = '1100';
            when '0101'  Toutput<4*i+3:4*i> = '0000';
            when '0110'  Toutput<4*i+3:4*i> = '1001';
            when '0111'  Toutput<4*i+3:4*i> = '1110';
            when '1000'  Toutput<4*i+3:4*i> = '0011';
            when '1001'  Toutput<4*i+3:4*i> = '0111';
            when '1010'  Toutput<4*i+3:4*i> = '0100';
            when '1011'  Toutput<4*i+3:4*i> = '0101';
            when '1100'  Toutput<4*i+3:4*i> = '1101';
            when '1101'  Toutput<4*i+3:4*i> = '0010';
            when '1110'  Toutput<4*i+3:4*i> = '0001';
            when '1111'  Toutput<4*i+3:4*i> = '1010';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =========

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =================

bits(4) TweakCellInvRot(bits(4)incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =================

bits(64) TweakInvShuffle(bits(64)indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// ==============

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
```

Library pseudocode for aarch64/functions/pac/pac/HavePACExt

```
// HavePACExt()
// ============

boolean HavePACExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// ==============================

// Returns TRUE if the pointer has upper and lower address ranges

boolean PtrHasUpperAndLowerAddRanges()
    return PSTATE.EL == EL1 || PSTATE.EL == EL0 || (PSTATE.EL == EL2 && HCR_EL2.E2H == '1');
```

Library pseudocode for aarch64/functions/pac/strip/Strip

```
// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = CalculateTBI(A, data);
    integer bottom_PAC_bit = CalculateBottomPACBit(A, A<55>);
    extfield = Replicate(A<55>, 64);

    if tbi then
        original_ptr = A<63:56>:extfield< 56-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield< 64-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;

    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = !HaveEL(EL2) || HCR_EL2.TGE=='0' || HCR_EL2.E2H=='0';
            TrapEL2 = HaveEL(EL2) && IsEL1Regime && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = HaveEL(EL2) && !IsSecure() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else  return original_ptr;
```

Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el !=  EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

```
// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()

    route_to_el3 = (HaveEL(EL3) && SCR_EL3.EA == '1');
    route_to_el2 = (HaveEL(EL2) && !IsSecure() &&
                    (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');
    intdis      = (Halted() || ExternalDebugInterruptsDisabled(target));
    masked      = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending
    if SErrorPending() && masked then       // ISR_EL1.A is set to 1
        // 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 HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1};

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = (HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1');
    vSEI_pending = (vSEI_enabled && HCR_EL2.VSE == '1');
    vintdis      = (Halted() || ExternalDebugInterruptsDisabled(EL1));
    vmasked      = (vintdis || PSTATE.A == '1');

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if ELUsingAArch32(EL1) then
            VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        else
            VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>);
        HCR_EL2.VSE = '0';                        // Clear pending virtual SError

    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} && HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[n]<63:32> = Zeros();

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// ===============================

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i] = bits(64) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// ==============================

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// ===============================

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;
    ELR_EL1  = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(32) UNKNOWN;
        ELR_EL2  = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(32) UNKNOWN;
        ELR_EL3  = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq = bits(32) UNKNOWN;
        SPSR_irq = bits(32) UNKNOWN;
        SPSR_abt = bits(32) UNKNOWN;
        SPSR_und = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSystemRegisters

```
AArch64.ResetSystemRegisters(boolean cold_reset);
```

Library pseudocode for aarch64/functions/registers/PC

```
// PC - non-assignment form
// ========================
// Read program counter.

bits(64) PC[]
    return _PC;
```

Library pseudocode for aarch64/functions/registers/SP

```
// SP[] - assignment form
// ======================
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
    assert width IN {32,64};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value);
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = ZeroExtend(value);
            when EL1  SP_EL1 = ZeroExtend(value);
            when EL2  SP_EL2 = ZeroExtend(value);
            when EL3  SP_EL3 = ZeroExtend(value);
    return;

// SP[] - non-assignment form
// ==========================
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
    assert width IN {8,16,32,64};
    if PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0<width-1:0>;
            when EL1  return SP_EL1<width-1:0>;
            when EL2  return SP_EL2<width-1:0>;
            when EL3  return SP_EL3<width-1:0>;
```

Library pseudocode for aarch64/functions/registers/V

```
// V[] - assignment form
// =====================
// 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};
    _V[n] = ZeroExtend(value);
    return;

// 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 _V[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 IN {8,16,32,64};
        return _V[n]<width-1:0>;
    else
        assert width == 64;
        return _V[n]<(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 IN {8,16,32,64};
        _V[n] = ZeroExtend(value);
    else
        assert width == 64;
        _V[n]<(width * 2)-1:width> = value<width-1:0>;
```

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/sysregisters/CNTKCTL

```
// CNTKCTL[] - non-assignment form
// ===============================

CNTKCTLType CNTKCTL[]
    if IsInHost() then
        return CNTHCTL_EL2;
    return CNTKCTL_EL1;
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

Library pseudocode for aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// =============================

CPACRType CPACR[]
    if IsInHost() then
        return CPTR_EL2;
    return CPACR_EL1;
```

Library pseudocode for aarch64/functions/sysregisters/CPACRType

```
type CPACRType;
```

Library pseudocode for aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// ===========================

bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        when EL3  r = ELR_EL3;
        otherwise Unreachable();
    return r;

// ELR[] - non-assignment form
// ===========================

bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// =======================

ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1  ELR_EL1 = r;
        when EL2  ELR_EL2 = r;
        when EL3  ELR_EL3 = r;
        otherwise Unreachable();
    return;

// ELR[] - assignment form
// =======================

ELR[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR[PSTATE.EL] = value;
    return;
```

Library pseudocode for aarch64/functions/sysregisters/ESR

```
// ESR[] - non-assignment form
// ===========================

ESRType ESR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        when EL3  r = ESR_EL3;
        otherwise Unreachable();
    return r;

// ESR[] - non-assignment form
// ===========================

ESRType ESR[]
    return ESR[S1TranslationRegime()];

// ESR[] - assignment form
// =======================

ESR[bits(2) regime] = ESRType value
    bits(32) r = value;
    case regime of
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        when EL3  ESR_EL3 = r;
        otherwise Unreachable();
    return;

// ESR[] - assignment form
// =======================

ESR[] = ESRType value
    ESR[S1TranslationRegime()] = value;
```

Library pseudocode for aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

Library pseudocode for aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// ===========================

bits(64) FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = FAR_EL1;
        when EL2  r = FAR_EL2;
        when EL3  r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR[] - non-assignment form
// ===========================

bits(64) FAR[]
    return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// =======================

FAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1  FAR_EL1 = r;
        when EL2  FAR_EL2 = r;
        when EL3  FAR_EL3 = r;
        otherwise Unreachable();
    return;

// FAR[] - assignment form
// =======================

FAR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;
```

Library pseudocode for aarch64/functions/sysregisters/MAIR

```
// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = MAIR_EL1;
        when EL2  r = MAIR_EL2;
        when EL3  r = MAIR_EL3;
        otherwise Unreachable();
    return r;

// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[]
    return MAIR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

Library pseudocode for aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// ============================

SCTLRType SCTLR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        when EL3  r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// ============================

SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

Library pseudocode for aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/system/AArch64.CheckAdvSIMDFPSystemRegisterTraps

```
// Checks if an AArch64 MSR, MRS or SYS instruction on a SIMD or floating-point
// register is trapped under the current configuration. Returns a boolean which
// is TRUE if trapping occurs, plus a binary value that specifies the Exception
// level trapped to.
(boolean, bits(2)) AArch64.CheckAdvSIMDFPSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bit
```

Library pseudocode for aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// ===========================

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bits(5) rt,
    // Checks if an AArch64 MSR, MRS or SYS instruction is UNALLOCATED or trapped at the current
    // exception level, security state and configuration, based on the opcode's encoding.
    boolean unallocated = FALSE;
    boolean need_secure = FALSE;
    bits(2) min_EL;

    // Check for traps by HCR_EL2.TIDCP
    if HaveEL(EL2) && !IsSecure() && HCR_EL2.TIDCP == 1 && op0 == 'x1' && crn == '1x11' then
        // At Non-secure EL0, it is IMPLEMENTATION_DEFINED whether attempts to execute system
        // register access instructions with reserved encodings are trapped to EL2 or UNDEFINED
        rcs_el0_trap = boolean IMPLEMENTATION_DEFINED "Reserved Control Space EL0 Trapped";
        if PSTATE.EL == EL0 && rcs_el0_trap then
            AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);
        elsif PSTATE.EL == EL1 then
            AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);

    // Check for unallocated encodings
    case op1 of
        when '00x', '010'
            min_EL = EL1;
        when '011'
            min_EL = EL0;
        when '100'
            min_EL = EL2;
        when '101'
            if !HaveVirtHostExt() then UnallocatedEncoding();
            min_EL = EL2;
        when '110'
            min_EL = EL3;
        when '111'
            min_EL = EL1;
            need_secure = TRUE;

    if UInt(PSTATE.EL) < UInt(min_EL) then
        // Check for traps on read/write access to registers named _EL2, _EL02, _EL12 from non-secure E
        if ( PSTATE.EL == EL1 && min_EL == EL2 && HaveNVExt() && !IsSecure() && HaveEL(EL2) && HCR_EL2.
            AArch64.SystemRegisterTrap(EL2, op0, op2, op1, crn, rt, crm, read);
        else
            UnallocatedEncoding();
    elsif need_secure && !IsSecure() then
        UnallocatedEncoding();
    elsif AArch64.CheckUnallocatedSystemAccess(op0, op1, crn, crm, op2, read) then
        UnallocatedEncoding();

    // Check for traps on accesses to SIMD or floating-point registers
    (take_trap, target_el) = AArch64.CheckAdvSIMDFPSystemRegisterTraps(op0, op1, crn, crm, op2, read);
    if take_trap then
        AArch64.AdvSIMDFPAccessTrap(target_el);

    // Check for traps on access to all other system registers
    (take_trap, target_el) = AArch64.CheckSystemRegisterTraps(op0, op1, crn, crm, op2, read);

    if take_trap then
        AArch64.SystemRegisterTrap(target_el, op0, op2, op1, crn, rt, crm, read);
```

Library pseudocode for aarch64/functions/system/AArch64.CheckSystemRegisterTraps

```
// Checks if an AArch64 MSR, MRS or SYS instruction on a system register is trapped
// under the current configuration. Returns a boolean which is TRUE if trapping
// occurs, plus a binary value that specifies the Exception level trapped to.
(boolean, bits(2)) AArch64.CheckSystemRegisterTraps(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
```

Library pseudocode for aarch64/functions/system/AArch64.CheckUnallocatedSystemAccess

```
// Checks if an AArch64 MSR, MRS or SYS instruction is unallocated under the current
// configuration.
boolean AArch64.CheckUnallocatedSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingATS1xPInstr

```
// AArch64.ExecutingATS1xPInstr()
// ==============================
// Return TRUE if current instruction is AT S1E1R/WP

boolean AArch64.ExecutingATS1xPInstr()
    if !HavePrivATExt() then return FALSE;

    instr = ThisInstr();
    if instr<22+:10> == '1101010100' then
        op1  = instr<16+:3>;
        CRn  = instr<12+:4>;
        CRm  = instr<8+:4>;
        op2  = instr<5+:3>;
        return (op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000','001'});
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/AArch64.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) System_Get(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/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)

    SynchronizeContext();

    if HaveRASExt() && SCTLR[].IESB == '1' then
        ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then                // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set s
        if PSTATE.T == '0' then
            new_pc<0> = '0';                 // T32
        else
            new_pc<1:0> = '00';              // A32
    else                                     // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc);

    if UsingAArch32() then
        // 32 most significant bits are ignored.
        BranchTo(new_pc<31:0>, BranchType_UNKNOWN);
    else
        BranchToAddr(new_pc, BranchType_ERET);
```

Library pseudocode for aarch64/instrs/countop/CountOp

```
enumeration CountOp     {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

Library pseudocode for aarch64/instrs/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =================
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```

Library pseudocode for aarch64/instrs/extendreg/ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType type, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;

    case type of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SXTX 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_UXTX 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_SXTX,
                         ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

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;
```

Library pseudocode for aarch64/instrs/integer/bitmasks/DecodeBitMasks

```
// 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 ReservedValue();
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        ReservedValue();

    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 equivalant to:
    //   esize = 1 << len;
    //   d = UInt(diff<len-1:0>);
    //   welem = ZeroExtend(Ones(S + 1), esize);
    //   telem = ZeroExtend(Ones(d + 1), esize);
    //   wmask = Replicate(ROR(welem, R));
    //   tmask = Replicate(telem);
    //   return (wmask, tmask);

    // Compute "top mask"
    tmask_and = diff<5:0> OR NOT(levels);
    tmask_or  = diff<5:0> AND levels;

    tmask = Ones(64);
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
               OR  Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
               OR  Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
               OR  Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
               OR  Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
               OR  Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
               OR  Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

    // Compute "wraparound mask"
    wmask_and = immr OR NOT(levels);
    wmask_or  = immr AND levels;

    wmask = Zeros(64);
    wmask = ((wmask
```

```
                AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
                 OR  Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
    // optimization of first step:
    // wmask = Replicate(wmask_or<0> : '0', 32);
    wmask = ((wmask
                AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
                 OR  Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
    wmask = ((wmask
                AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
                 OR  Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
    wmask = ((wmask
                AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
                 OR  Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
    wmask = ((wmask
                AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
                 OR  Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
    wmask = ((wmask
                AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
                 OR  Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

    if diff<6> != '0' then // borrow from S - R
        wmask = wmask AND tmask;
    else
        wmask = wmask OR tmask;

    return (wmask<M-1:0>, tmask<M-1:0>);
```

Library pseudocode for aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp

```
enumeration MoveWideOp  {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

Library pseudocode for aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// ===================
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && immN:imms != '1xxxxxx' then
        return FALSE;
    if sf == '0' && immN:imms != '00xxxxx' then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if S < 16 then
        // ones must not span halfword boundary when rotated
        return (-R MOD 16) <= (15 - S);

    // for MOVN must contain no more than 16 zeros
    if S >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (R MOD 16) <= (S - (width - 15));

    return FALSE;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/DecodeShift

```
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00'  return ShiftType_LSL;
        when '01'  return ShiftType_LSR;
        when '10'  return ShiftType_ASR;
        when '11'  return ShiftType_ROR;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftReg

```
// ShiftReg()
// ==========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType type, integer amount)
    bits(N) result = X[reg];
    case type of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftType

```
enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

Library pseudocode for aarch64/instrs/logicalop/LogicalOp

```
enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

Library pseudocode for aarch64/instrs/memory/memop/MemAtomicOp

```
enumeration MemAtomicOp {MemAtomicOp_ADD,
                         MemAtomicOp_BIC,
                         MemAtomicOp_EOR,
                         MemAtomicOp_ORR,
                         MemAtomicOp_SMAX,
                         MemAtomicOp_SMIN,
                         MemAtomicOp_UMAX,
                         MemAtomicOp_UMIN,
                         MemAtomicOp_SWP};
```

Library pseudocode for aarch64/instrs/memory/memop/MemOp

```
enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

Library pseudocode for aarch64/instrs/memory/prefetch/Prefetch

```
// Prefetch()
// ==========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;         // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;         // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;        // PST: prepare for store
        when '11' return;                       // unallocated hint
    target = UInt(prfop<2:1>);                  // target cache level
    stream = (prfop<0> != '0');                 // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

Library pseudocode for aarch64/instrs/system/barriers/barrierop/MemBarrierOp

```
enumeration MemBarrierOp   {MemBarrierOp_DSB, MemBarrierOp_DMB, MemBarrierOp_ISB};
```

Library pseudocode for aarch64/instrs/system/hints/syshintop/SystemHintOp

```
enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
};
```

Library pseudocode for aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_PAN, // ARMv8.1
                         PSTATEField_UAO, // ARMv8.2
                         PSTATEField_SP
                         };
```

Library pseudocode for aarch64/instrs/system/sysops/sysop/SysOp

```
// SysOp()
// =======

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;   // S1E1R
        when '100 0111 1000 000' return Sys_AT;   // S1E2R
        when '110 0111 1000 000' return Sys_AT;   // S1E3R
        when '000 0111 1000 001' return Sys_AT;   // S1E1W
        when '100 0111 1000 001' return Sys_AT;   // S1E2W
        when '110 0111 1000 001' return Sys_AT;   // S1E3W
        when '000 0111 1000 010' return Sys_AT;   // S1E0R
        when '000 0111 1000 011' return Sys_AT;   // S1E0W
        when '100 0111 1000 100' return Sys_AT;   // S12E1R
        when '100 0111 1000 101' return Sys_AT;   // S12E1W
        when '100 0111 1000 110' return Sys_AT;   // S12E0R
        when '100 0111 1000 111' return Sys_AT;   // S12E0W
        when '011 0111 0100 001' return Sys_DC;   // ZVA
        when '000 0111 0110 001' return Sys_DC;   // IVAC
        when '000 0111 0110 010' return Sys_DC;   // ISW
        when '011 0111 1010 001' return Sys_DC;   // CVAC
        when '000 0111 1010 010' return Sys_DC;   // CSW
        when '011 0111 1011 001' return Sys_DC;   // CVAU
        when '011 0111 1110 001' return Sys_DC;   // CIVAC
        when '000 0111 1110 010' return Sys_DC;   // CISW
        when '000 0111 0001 000' return Sys_IC;   // IALLUIS
        when '000 0111 0101 000' return Sys_IC;   // IALLU
        when '011 0111 0101 001' return Sys_IC;   // IVAU
        when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
        when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
        when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
        when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
        when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
        when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
        when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
        when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
        when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
        when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
        when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
        when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
        when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
        when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
        when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
        when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
        when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
        when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
        when '100 1000 0111 000' return Sys_TLBI; // ALLE2
        when '110 1000 0111 000' return Sys_TLBI; // ALLE3
        when '000 1000 0111 001' return Sys_TLBI; // VAE1
        when '100 1000 0111 001' return Sys_TLBI; // VAE2
        when '110 1000 0111 001' return Sys_TLBI; // VAE3
        when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
        when '000 1000 0111 011' return Sys_TLBI; // VAAE1
        when '100 1000 0111 100' return Sys_TLBI; // ALLE1
        when '000 1000 0111 101' return Sys_TLBI; // VALE1
        when '100 1000 0111 101' return Sys_TLBI; // VALE2
        when '110 1000 0111 101' return Sys_TLBI; // VALE3
        when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
        when '000 1000 0111 111' return Sys_TLBI; // VAALE1
    return Sys_SYS;
```

Library pseudocode for aarch64/instrs/system/sysops/sysop/SystemOp

```
enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```

Library pseudocode for aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```
enumeration VBitOp      {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

Library pseudocode for aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```
enumeration CompareOp   {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                         CompareOp_LE, CompareOp_LT};
```

Library pseudocode for aarch64/instrs/vector/crypto/enabled/CheckCryptoEnabled64

```
// CheckCryptoEnabled64()
// ======================

CheckCryptoEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
    return;
```

Library pseudocode for aarch64/instrs/vector/logical/immediateop/ImmediateOp

```
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                         ImmediateOp_ORR, ImmediateOp_BIC};
```

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/Reduce

```
// Reduce()
// ========

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize);
    lo = Reduce(op, input<half-1:0>, esize);

    case op of
        when ReduceOp_FMINNUM
            result = FPMinNum(lo, hi, FPCR);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, FPCR);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, FPCR);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, FPCR);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR);
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/ReduceOp

```
enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                      ReduceOp_FMIN, ReduceOp_FMAX,
                      ReduceOp_FADD, ReduceOp_ADD};
```

Library pseudocode for aarch64/translation/attrs/AArch64.InstructionDevice

```
// AArch64.InstructionDevice()
// ===========================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch64.InstructionDevice(AddressDescriptor addrdesc, bits(64) vaddress,
                                            bits(52) ipaddress, integer level,
                                            AccType acctype, boolean iswrite, boolean secondstage,
                                            boolean s2fs1walk)

    c = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch64.PermissionFault(ipaddress, level, acctype, iswrite,
                                                 secondstage, s2fs1walk);
    else
        addrdesc.memattrs.type = MemType_Normal;
        addrdesc.memattrs.inner.attrs = MemAttr_NC;
        addrdesc.memattrs.inner.hints = MemHint_No;
        addrdesc.memattrs.outer = addrdesc.memattrs.inner;
        addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

    return addrdesc;
```

Library pseudocode for aarch64/translation/attrs/AArch64.S1AttrDecode

```
// AArch64.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch64.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;

    mair = MAIR[];
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);

    if attrfield<7:4> == '0000' then            // Device
        memattrs.type = MemType_Device;
        case attrfield<3:0> of
            when '0000'  memattrs.device = DeviceType_nGnRnE;
            when '0100'  memattrs.device = DeviceType_nGnRE;
            when '1000'  memattrs.device = DeviceType_nGRE;
            when '1100'  memattrs.device = DeviceType_GRE;
            otherwise    Unreachable();         // Reserved, handled above

    elsif attrfield<3:0> != '0000'  then        // Normal
        memattrs.type = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';

    else
        Unreachable();                          // Reserved, handled above

    return MemAttrDefaults(memattrs);
```

Library pseudocode for aarch64/translation/attrs/AArch64.TranslateAddressS1Off

```
// AArch64.TranslateAddressS1Off()
// ===============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch64.TranslateAddressS1Off(bits(64) vaddress, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    Top = AddrTop(vaddress, FALSE, PSTATE.EL);
    if !IsZero(vaddress<Top:PAMax()>) then
        level = 0;
        ipaddress = bits(52) UNKNOWN;
        secondstage = FALSE;
        s2fs1walk = FALSE;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    default_cacheable = (HasS2Translation() && HCR_EL2.DC == '1');

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.type = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;      // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
    elsif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.type = MemType_Device;
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
    else
        // Instruction cacheability controlled by SCTLR_ELx.I
        cacheable = SCTLR[].I == '1';
        result.addrdesc.memattrs.type = MemType_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            result.addrdesc.memattrs.inner.hints = MemHint_No;
        result.addrdesc.memattrs.shareable = TRUE;
        result.addrdesc.memattrs.outershareable = TRUE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;

    result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.physicaladdress = vaddress<51:0>;
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch64.NoFault();
    return result;
```

Library pseudocode for aarch64/translation/checks/AArch64.AccessIsPrivileged

```
// AArch64.AccessIsPrivileged()
// ============================

boolean AArch64.AccessIsPrivileged(AccType acctype)

    if PSTATE.EL == EL0 then
        ispriv = FALSE;
    elsif PSTATE.EL == EL3 then
        ispriv = TRUE;
    elsif PSTATE.EL == EL2 && (!IsInHost() || HCR_EL2.TGE == '0') then
        ispriv = TRUE;
    elsif HaveUAOExt() && PSTATE.UAO == '1' then
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType_UNPRIV);

    return ispriv;
```

Library pseudocode for aarch64/translation/checks/AArch64.CheckPermission

```
// AArch64.CheckPermission()
// =========================
// Function used for permission checking from AArch64 stage 1 translations

FaultRecord AArch64.CheckPermission(Permissions perms, bits(64) vaddress, integer level,
                                    bit NS, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    wxn = SCTLR[].WXN == '1';

    if PSTATE.EL IN {EL0,EL1} || IsInHost() then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';

        ispriv = AArch64.AccessIsPrivileged(acctype);

        pan = if HavePANExt() then PSTATE.PAN else '0';
        if (HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1 && !IsSecure() && PSTATE.
            pan = '0';
        if (pan == '1' && user_r && ispriv &&
            !(acctype IN {AccType_DC,AccType_AT,AccType_IFETCH}) ||
            (acctype == AccType_AT && AArch64.ExecutingATS1xPInstr())) then
            priv_r = FALSE;
            priv_w = FALSE;

        user_xn = perms.xn == '1' || (user_w && wxn);
        priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2 or EL3
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
        xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW } then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(52) UNKNOWN;
        return AArch64.PermissionFault(ipaddress, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

Library pseudocode for aarch64/translation/checks/AArch64.CheckS2Permission

```
// AArch64.CheckS2Permission()
// ===========================
// Function used for permission checking from AArch64 stage 2 translations

FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, bits(52) ipaddress,
                                      integer level, AccType acctype, boolean iswrite,
                                      boolean s2fs1walk, boolean hwupdatewalk)

    assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HasS2Translation();

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    if HaveExtendedExecuteNeverExt() then
        case perms.xn:perms.xxn of
            when '00'  xn = FALSE;
            when '01'  xn = PSTATE.EL == EL1;
            when '10'  xn = TRUE;
            when '11'  xn = PSTATE.EL == EL0;
    else
        xn = perms.xn == '1';
    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
        failedread = TRUE;
    elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW }) && !s2fs1walk then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    elsif hwupdatewalk then
        fail = !w;
        failedread = !iswrite;
    else
        fail = !r;
        failedread = !iswrite;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch64.PermissionFault(ipaddress, level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime.
// The breakpoint can in fact be evaluated well ahead of execution, for example, at instruction
// fetch. This is the simple sequential execution of the program.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, 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, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch64.NoFault();

    d_side = (acctype != AccType_IFETCH);
    generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch64.CheckBreakpoint(vaddress, size);

    return fault;
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address".

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = AArch64.AccessIsPrivileged(acctype);

    for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
        match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elsif match && MDSCR_EL1.MDE == '1' && AArch64.GenerateDebugExceptions() then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

Library pseudocode for aarch64/translation/faults/AArch64.AccessFlagFault

```
// AArch64.AccessFlagFault()
// =========================

FaultRecord AArch64.AccessFlagFault(bits(52) ipaddress, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.AddressSizeFault

```
// AArch64.AddressSizeFault()
// ==========================

FaultRecord AArch64.AddressSizeFault(bits(52) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.AlignmentFault

```
// AArch64.AlignmentFault()
// ========================

FaultRecord AArch64.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch64.CreateFaultRecord(Fault_Alignment, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.AsynchExternalAbort

```
// AArch64.AsynchExternalAbort()
// ============================
// Wrapper function for asynchronous external aborts

FaultRecord AArch64.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

    type = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(type, ipaddress,  level, acctype, iswrite, extflag,
                                     errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// ====================

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)

    ipaddress = bits(52) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_Debug, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.NoFault

```
// AArch64.NoFault()
// =================

FaultRecord AArch64.NoFault()

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_None, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.PermissionFault

```
// AArch64.PermissionFault()
// =========================

FaultRecord AArch64.PermissionFault(bits(52) ipaddress, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Permission, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/faults/AArch64.TranslationFault

```
// AArch64.TranslationFault()
// ==========================

FaultRecord AArch64.TranslationFault(bits(52) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Translation, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

Library pseudocode for aarch64/translation/translation/AArch64.CheckAndUpdateDescriptor

```
// AArch64.CheckAndUpdateDescriptor()
// ==================================
// Check and update translation table descriptor if hardware update is configured

FaultRecord AArch64.CheckAndUpdateDescriptor(DescriptorUpdate result, FaultRecord fault,
                                             boolean secondstage, bits(64) vaddress, AccType acctype,
                                             boolean iswrite, boolean s2fs1walk, boolean hwupdatewalk)

    // Check if access flag can be updated
    // Address translation instructions are permitted to update AF but not required
    if result.AF then
        if fault.type == Fault_None then
            hw_update_AF = TRUE;
        elsif ConstrainUnpredictable(Unpredictable_AFUPDATE) == Constraint_TRUE then
            hw_update_AF = TRUE;
        else
            hw_update_AF = FALSE;

    if result.AP && fault.type == Fault_None then
        write_perm_req = (iswrite || acctype IN {AccType_ATOMICRW,AccType_ORDEREDRW}) && !s2fs1walk;
        hw_update_AP = (write_perm_req && !(acctype IN {AccType_AT,AccType_DC})) || hwupdatewalk;
    else
        hw_update_AP = FALSE;

    if hw_update_AF || hw_update_AP then
        if secondstage || !HasS2Translation() then
            descaddr2 = result.descaddr;
        else
            hwupdatewalk = TRUE;
            descaddr2 = AArch64.SecondStageWalk(result.descaddr, vaddress, acctype, iswrite, 8, hwupdat
            if IsFault(descaddr2) then
                return descaddr2.fault;

        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        desc = _Mem[descaddr2, 8, accdesc];

        if hw_update_AF then
            desc<10> = '1';
        if hw_update_AP then
            desc<7> = (if secondstage then '1' else '0');

        _Mem[descaddr2,8,accdesc] = desc;

    return fault;
```

Library pseudocode for aarch64/translation/translation/AArch64.FirstStageTranslate

```
// AArch64.FirstStageTranslate()
// =============================
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.FirstStageTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                              boolean wasaligned, integer size)

    if HasS2Translation() then
        s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';
    else
        s1_enabled = SCTLR[].M == '1';

    ipaddress = bits(52) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                          // First stage enabled
        S1 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);
        permissioncheck = TRUE;
    else
        S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;

        if UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() then
            if S1.addrdesc.memattrs.type == MemType_Device && S1.addrdesc.memattrs.device != DeviceType
                nTLSMD = if S1TranslationRegime() == EL2 then SCTLR_EL2.nTLSMD else SCTLR_EL1.nTLSMD;
                if nTLSMD == '0' then
                    S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
    // Check for unaligned data accesses to Device memory
    if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
        && S1.addrdesc.memattrs.type == MemType_Device && !IsFault(S1.addrdesc) then
        S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level,
                                                    S1.addrdesc.paddress.NS,
                                                    acctype, iswrite);

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.type == MemType_Device &&
        acctype == AccType_IFETCH) then
        S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                                acctype, iswrite,
                                                secondstage, s2fs1walk);

    // Check and update translation table descriptor if required
    hwupdatewalk = FALSE;
    s2fs1walk = FALSE;
    S1.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S1.descupdate, S1.addrdesc.fault,
                                                         secondstage, vaddress, acctype,
                                                         iswrite, s2fs1walk, hwupdatewalk);

    return S1.addrdesc;
```

Library pseudocode for aarch64/translation/translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch64.FullTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch64.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(S1) && HasS2Translation() then
        s2fs1walk = FALSE;
        hwupdatewalk = FALSE;
        result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                              size, hwupdatewalk);
    else
        result = S1;

    return result;
```

Library pseudocode for aarch64/translation/translation/AArch64.SecondStageTranslate

```
// AArch64.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.SecondStageTranslate(AddressDescriptor S1, bits(64) vaddress,
                                               AccType acctype, boolean iswrite, boolean wasaligned,
                                               boolean s2fs1walk, integer size, boolean hwupdatewalk)
    assert HasS2Translation();

    s2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';
    secondstage = TRUE;

    if s2_enabled then                        // Second stage enabled
        ipaddress = S1.paddress.physicaladdress<51:0>;

        S2 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                          s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
            && S2.addrdesc.memattrs.type == MemType_Device && !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                          acctype, iswrite, s2fs1walk, hwupdatewalk);
        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.type == MemType_Device &&
            acctype == AccType_IFETCH) then
            S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    acctype, iswrite,
                                                    secondstage, s2fs1walk);

        // Check for protected table walk
        if (s2fs1walk && !IsFault(S2.addrdesc) && HCR_EL2.PTW == '1' &&
            S2.addrdesc.memattrs.type == MemType_Device) then
            S2.addrdesc.fault = AArch64.PermissionFault(ipaddress, S2.level, acctype,
                                                        iswrite, secondstage, s2fs1walk);

        // Check and update translation table descriptor if required
        S2.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S2.descupdate, S2.addrdesc.fault,
                                                             secondstage, vaddress, acctype,
                                                             iswrite, s2fs1walk, hwupdatewalk);
        result = CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

    return result;
```

Library pseudocode for aarch64/translation/translation/AArch64.SecondStageWalk

```
// AArch64.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype,
                                          boolean iswrite, integer size, boolean hwupdatewalk)

    assert HasS2Translation();

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                        size, hwupdatewalk);
```

Library pseudocode for aarch64/translation/translation/AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)

    result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);

    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
```

Library pseudocode for aarch64/translation/translation/IsEL1TransRegimeRegs

```
// IsEL1TransRegimeRegs()
// ======================

// Returns TRUE if its a register in EL1 translation regime

boolean IsEL1TransRegimeRegs()
    return !HaveEL(EL2) || PSTATE.EL == EL1 || (PSTATE.EL == EL0 && (HCR_EL2.E2H == '0' ||  HCR_EL2.TGE
```

Library pseudocode for aarch64/translation/walk/AArch64.TranslationTableWalk

```
// AArch64.TranslationTableWalk()
// ==============================
// Returns a result of a translation table walk
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch64.TranslationTableWalk(bits(52) ipaddress, bits(64) vaddress,
                                       AccType acctype, boolean iswrite, boolean secondstage,
                                       boolean s2fs1walk, integer size)
    if !secondstage then
        assert !ELUsingAArch32(S1TranslationRegime());
    else
        assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HasS2Translation();

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(64) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2

    descaddr.memattrs.type = MemType_Normal;

    // Derived parameters for the page table walk:
    //  grainsize = Log2(Size of Table)         - Size of Table is 4KB, 16KB or 64KB in AArch64
    //  stride = Log2(Address per Level)        - Bits of address consumed at each level
    //  firstblocklevel = First level where a block entry is allowed
    //  ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTCR_EL2.PS
    //  inputsize = Log2(Size of Input Address) - Input Address size in bits
    //  level = Level to start walk from
    // This means that the number of levels after start level = 3-level

    if !secondstage then
        // First stage translation
        inputaddr = ZeroExtend(vaddress);
        top = AddrTop(inputaddr, (acctype == AccType_IFETCH), PSTATE.EL);
        if PSTATE.EL == EL3 then
            largegrain = TCR_EL3.TG0 == '01';
            midgrain = TCR_EL3.TG0 == '10';
            inputsize = 64 - UInt(TCR_EL3.T0SZ);
            inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
            if inputsize > inputsize_max then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - 39;
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            ps = TCR_EL3.PS;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<to
            disabled = FALSE;
            baseregister = TTBR0_EL3;
            descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0, secondstage);
            reversedescriptors = SCTLR_EL3.EE == '1';
            lookupsecure = TRUE;
            singlepriv = TRUE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL3.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL3.HD == '1';
            hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL3.HPD == '1';
        elsif IsInHost() then
            if inputaddr<top> == '0' then
                largegrain = TCR_EL2.TG0 == '01';
                midgrain = TCR_EL2.TG0 == '10';
                inputsize = 64 - UInt(TCR_EL2.T0SZ);
                inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
                if inputsize > inputsize_max then
                    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
```

```
            if c == Constraint_FORCE then inputsize = inputsize_max;
        inputsize_min = 64 - 39;
        if inputsize < inputsize_min then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_min;
        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputadd
        disabled = TCR_EL2.EPD0 == '1';
        baseregister = TTBR0_EL2;
        descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondsta
        hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD0 == '1';
    else
        inputsize = 64 - UInt(TCR_EL2.T1SZ);
        largegrain = TCR_EL2.TG1 == '11';       // TG1 and TG0 encodings differ
        midgrain = TCR_EL2.TG1 == '01';
        inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
        if inputsize > inputsize_max then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_max;
        inputsize_min = 64 - 39;
        if inputsize < inputsize_min then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_min;
        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputadd
        disabled = TCR_EL2.EPD1 == '1';
        baseregister = TTBR1_EL2;
        descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH1, TCR_EL2.ORGN1, TCR_EL2.IRGN1, secondsta
        hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD1 == '1';
    ps = TCR_EL2.IPS;
    reversedescriptors = SCTLR_EL2.EE == '1';
    lookupsecure = FALSE;
    singlepriv = FALSE;
    update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
    update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
elsif PSTATE.EL == EL2 then
    inputsize = 64 - UInt(TCR_EL2.T0SZ);
    largegrain = TCR_EL2.TG0 == '01';
    midgrain = TCR_EL2.TG0 == '10';
    inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
    if inputsize > inputsize_max then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_max;
    inputsize_min = 64 - 39;
    if inputsize < inputsize_min then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_min;
    ps = TCR_EL2.PS;
    basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<to
    disabled = FALSE;
    baseregister = TTBR0_EL2;
    descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondstage);
    reversedescriptors = SCTLR_EL2.EE == '1';
    lookupsecure = FALSE;
    singlepriv = TRUE;
    update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
    update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
    hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD == '1';
else
    if inputaddr<top> == '0' then
        inputsize = 64 - UInt(TCR_EL1.T0SZ);
        largegrain = TCR_EL1.TG0 == '01';
        midgrain = TCR_EL1.TG0 == '10';
        inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
        if inputsize > inputsize_max then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
```

```
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - 39;
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputadd
            disabled = TCR_EL1.EPD0 == '1';
            baseregister = TTBR0_EL1;
            descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORGN0, TCR_EL1.IRGN0, secondsta
            hierattrsdisabled = AArch64.HaveHPDExt() &&  TCR_EL1.HPD0 == '1';
        else
            inputsize = 64 - UInt(TCR_EL1.T1SZ);
            largegrain = TCR_EL1.TG1 == '11';       // TG1 and TG0 encodings differ
            midgrain = TCR_EL1.TG1 == '01';
            inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
            if inputsize > inputsize_max then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_max;
            inputsize_min = 64 - 39;
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputadd
            disabled = TCR_EL1.EPD1 == '1';
            baseregister = TTBR1_EL1;
            descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGN1, TCR_EL1.IRGN1, secondsta
            hierattrsdisabled = AArch64.HaveHPDExt() &&  TCR_EL1.HPD1 == '1';
        ps = TCR_EL1.IPS;
        reversedescriptors = SCTLR_EL1.EE == '1';
        lookupsecure = IsSecure();
        singlepriv = FALSE;
        update_AF = HaveAccessFlagUpdateExt() && TCR_EL1.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL1.HD == '1';
    if largegrain then
        grainsize = 16;                                         // Log2(64KB page size)
        firstblocklevel = (if Have52BitPAExt() then 1 else 2);  // Largest block is 4TB (2^42 b
                                                                // and 512MB (2^29 bytes) other
    elsif midgrain then
        grainsize = 14;                                         // Log2(16KB page size)
        firstblocklevel = 2;                                    // Largest block is 32MB (2^25
    else // Small grain
        grainsize = 12;                                         // Log2(4KB page size)
        firstblocklevel = 1;                                    // Largest block is 1GB (2^30 b
    stride = grainsize - 3;                                     // Log2(page size / 8 bytes)
    // The starting level is the number of strides needed to consume the input address
    level = 4 - RoundUp(Real(inputsize - grainsize) / Real(stride));

else
    // Second stage translation
    inputaddr = ZeroExtend(ipaddress);
    inputsize = 64 - UInt(VTCR_EL2.T0SZ);
    largegrain = VTCR_EL2.TG0 == '01';
    midgrain = VTCR_EL2.TG0 == '10';
    inputsize_max = (if Have52BitVAExt() && largegrain then 52 else 48);
    if inputsize > inputsize_max then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_max;
    inputsize_min = 64 - 39;
    if inputsize < inputsize_min then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_min;
    ps = VTCR_EL2.PS;
    basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<63:inp
    disabled = FALSE;
    baseregister = VTTBR_EL2;
```

```
    descaddr.memattrs = WalkAttrDecode(VTCR_EL2.IRGN0, VTCR_EL2.ORGN0, VTCR_EL2.SH0, secondstage);
    reversedescriptors = SCTLR_EL2.EE == '1';
    lookupsecure = FALSE;
    singlepriv = TRUE;
    update_AF = HaveAccessFlagUpdateExt() && VTCR_EL2.HA == '1';
    update_AP = HaveDirtyBitModifierExt() && update_AF && VTCR_EL2.HD == '1';

    startlevel = UInt(VTCR_EL2.SL0);
    if largegrain then
        grainsize = 16;                                         // Log2(64KB page size)
        level = 3 - startlevel;
        firstblocklevel = (if Have52BitPAExt() then 1 else 2);  // Largest block is 4TB (2^42 bytes
                                                                // and 512MB (2^29 bytes) otherwise
    elsif midgrain then
        grainsize = 14;                                         // Log2(16KB page size)
        level = 3 - startlevel;
        firstblocklevel = 2;                                    // Largest block is 32MB (2^25 byte
    else // Small grain
        grainsize = 12;                                         // Log2(4KB page size)
        level = 2 - startlevel;
        firstblocklevel = 1;                                    // Largest block is 1GB (2^30 bytes
    stride = grainsize - 3;                                     // Log2(page size / 8 bytes)

    // Limits on IPA controls based on implemented PA size. Level 0 is only
    // supported by small grain translations
    if largegrain then                              // 64KB pages
        // Level 1 only supported if implemented PA size is greater than 2^42 bytes
        if level == 0 || (level == 1 && PAMax() <= 42) then basefound = FALSE;
    elsif midgrain then                             // 16KB pages
        // Level 1 only supported if implemented PA size is greater than 2^40 bytes
        if level == 0 || (level == 1 && PAMax() <= 40) then basefound = FALSE;
    else                                            // Small grain, 4KB pages
        // Level 0 only supported if implemented PA size is greater than 2^42 bytes
        if level < 0 || (level == 0 && PAMax() <= 42) then basefound = FALSE;

    // If the inputsize exceeds the PAMax value, the behavior is CONSTRAINED UNPREDICTABLE
    inputsizecheck = inputsize;
    if inputsize > PAMax() && (!ELUsingAArch32(EL1) || inputsize > 40) then
        case ConstrainUnpredictable(Unpredictable_LARGEIPA) of
            when Constraint_FORCE
                // Restrict the inputsize to the PAMax value
                inputsize = PAMax();
                inputsizecheck = PAMax();
            when Constraint_FORCENOSLCHECK
                // As FORCE, except use the configured inputsize in the size checks below
                inputsize = PAMax();
            when Constraint_FAULT
                // Generate a translation fault
                basefound = FALSE;
            otherwise
                Unreachable();

    // Number of entries in the starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    startsizecheck = inputsizecheck - ((3 - level)*stride + grainsize); // Log2(Num of entries)

    // Check for starting level table with fewer than 2 entries or longer than 16 pages.
    // Lower bound check is:  startsizecheck < Log2(2 entries)
    // Upper bound check is:  startsizecheck > Log2(pagesize/8*16)
    if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

if !basefound || disabled then
    level = 0;            // AArch32 reports this as a level 1 fault
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype, iswrite,
                                                     secondstage, s2fs1walk);
    return result;

case ps of
    when '000'  outputsize = 32;
    when '001'  outputsize = 36;
```

```
        when '010'  outputsize = 40;
        when '011'  outputsize = 42;
        when '100'  outputsize = 44;
        when '101'  outputsize = 48;
        when '110'  outputsize = (if Have52BitPAExt() && largegrain then 52 else 48);
        otherwise   outputsize = 48;

    if outputsize > PAMax() then outputsize = PAMax();

    if outputsize < 48 && !IsZero(baseregister<47:outputsize>) then
        level = 0;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    // Bottom bound of the Base address is:
    //     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
    // Number of entries in starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
    if outputsize == 52 then
        z = (if baselowerbound < 6 then 6 else baselowerbound);
        baseaddress = baseregister<5:2>:baseregister<47:z>:Zeros(z);
    else
        baseaddress = ZeroExtend(baseregister<47:baselowerbound>:Zeros(baselowerbound));

    ns_table = if lookupsecure then '0' else '1';
    ap_table = '00';
    xn_table = '0';
    pxn_table = '0';

    addrselecttop = inputsize - 1;

    apply_nvnv1_effect = HaveNVExt() && HaveEL(EL2) && HCR_EL2.NV == 1 && HCR_EL2.NV1 == 1;

    repeat
        addrselectbottom = (3-level)*stride + grainsize;

        bits(52) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
        descaddr.paddress.physicaladdress = baseaddress OR index;
        descaddr.paddress.NS = ns_table;

        // If there are two stages of translation, then the first stage table walk addresses
        // are themselves subject to translation
        if secondstage || !HasS2Translation() then
            descaddr2 = descaddr;
        else
            hwupdatewalk = FALSE;
            descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
            // Check for a fault on the stage 2 walk
            if IsFault(descaddr2) then
                result.addrdesc.fault = descaddr2.fault;
                return result;

        // Update virtual address for abort functions
        descaddr2.vaddress = ZeroExtend(vaddress);

        accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
        desc = _Mem[descaddr2, 8, accdesc];

        if reversedescriptors then desc = BigEndianReverse(desc);

        if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
             // Fault (00), Reserved (10), or Block (01) at level 3
            result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        // Valid Block, Page, or Table entry
        if desc<1:0> == '01' || level == 3 then                 // Block (01) or Page (11)
```

```
            blocktranslate = TRUE;
        else                                                        // Table (11)
            if (outputsize < 52 && largegrain && !IsZero(desc<15:12>)) || (outputsize < 48 && !IsZero(d
                result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                                  iswrite, secondstage, s2fs1walk);
                return result;

            if outputsize == 52 then
                baseaddress = desc<15:12>:desc<47:grainsize>:Zeros(grainsize);
            else
                baseaddress = ZeroExtend(desc<47:grainsize>:Zeros(grainsize));
            if !secondstage then
                // Unpack the upper and lower table attributes
                ns_table    = ns_table    OR desc<63>;
            if !secondstage && !hierattrsdisabled then
                ap_table<1> = ap_table<1> OR desc<62>;       // read-only

                if apply_nvnv1_effect then
                    pxn_table   = pxn_table   OR desc<60>;
                else
                    xn_table    = xn_table    OR desc<60>;
                // pxn_table and ap_table[0] apply in EL1&0 or EL2&0 translation regimes
                if !singlepriv then
                    if !apply_nvnv1_effect then
                        pxn_table   = pxn_table   OR desc<59>;
                        ap_table<0> = ap_table<0> OR desc<61>;   // privileged

            level = level + 1;
            addrselecttop = addrselectbottom - 1;
            blocktranslate = FALSE;
    until blocktranslate;

    // Check block size is supported at this level
    if level < firstblocklevel then
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Check for misprogramming of the contiguous bit
    if largegrain then
        contiguousbitcheck = level == 2 && inputsize < 34;
    elsif midgrain then
        contiguousbitcheck = level == 2 && inputsize < 30;
    else
        contiguousbitcheck = level == 1 && inputsize < 34;

    if contiguousbitcheck && desc<52> == '1' then
        if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
            result.addrdesc.fault = AArch64.TranslationFault(ipaddress, level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

    // Check the output address is inside the supported range
    if (outputsize < 52 && largegrain && !IsZero(desc<15:12>)) || (outputsize < 48 && !IsZero(desc<47:c
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Unpack the descriptor into address and upper and lower block attributes
    if outputsize == 52 then
        outputaddress = desc<15:12>:desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
    else
        outputaddress = ZeroExtend(desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>);
    // Check Access Flag
    if desc<10> == '0' then
        if !update_AF then
            result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress, level, acctype,
                                                            iswrite, secondstage, s2fs1walk);
            return result;
        else
```

```
            result.descupdate.AF = TRUE;

    if update_AP && desc<51> == '1' then
        // If hw update of access permission field is configured consider AP[2] as '0' / S2AP[2] as '1'
        if !secondstage && desc<7> == '1' then
            desc<7> = '0';
            result.descupdate.AP = TRUE;
        elsif secondstage && desc<7> == '0' then
            desc<7> = '1';
            result.descupdate.AP = TRUE;

    // Required descriptor if AF or AP[2]/S2AP[2] needs update
    result.descupdate.descaddr = descaddr;

    if apply_nvnv1_effect then
        pxn = desc<54>;
        xn = '0';
    else
        xn = desc<54>;
        pxn = desc<53>;
    contiguousbit = desc<52>;
    nG = desc<11>;
    sh = desc<9:8>;
    if apply_nvnv1_effect then
        ap = desc<7>:'01';                                  // Bit[6] is treated as 0 regardless of v
    else
        ap = desc<7:6>:'1';
    memattr = desc<5:2>;                                    // AttrIndx and NS bit in stage 1

    result.domain = bits(4) UNKNOWN;                        // Domains not used
    result.level = level;
    result.blocksize = 2^((3-level)*stride + grainsize);

    // Stage 1 translation regimes also inherit attributes from the tables
    if !secondstage then
        result.perms.xn      = xn OR xn_table;
        result.perms.ap<2>   = ap<2> OR ap_table<1>;        // Force read-only
        // PXN, nG and AP[1] apply in EL1&0 or EL2&0 stage 1 translation regimes
        if !singlepriv then
            result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
            result.perms.pxn   = pxn OR pxn_table;
            // Pages from Non-secure tables are marked non-global in Secure EL1&0
            if IsSecure() then
                result.nG = nG OR ns_table;
            else
                result.nG = nG;
        else
            result.perms.ap<1> = '1';
            result.perms.pxn   = '0';
            result.nG          = '0';
        result.perms.ap<0>   = '1';
        result.addrdesc.memattrs = AArch64.S1AttrDecode(sh, memattr<2:0>, acctype);
        result.addrdesc.paddress.NS = memattr<3> OR ns_table;
    else
        result.perms.ap<2:1> = ap<2:1>;
        result.perms.ap<0>   = '1';
        result.perms.xn      = xn;
        if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
        result.perms.pxn     = '0';
        result.nG            = '0';
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
        result.addrdesc.paddress.NS = '1';

    result.addrdesc.paddress.physicaladdress = outputaddress;
    result.addrdesc.fault = AArch64.NoFault();
    result.contiguous = contiguousbit == '1';
    if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;
    return result;
```

Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';            // Clear TX underrun flag
    EDSCR.RXO = '0';            // Clear RX overrun flag

    if Halted() then            // in Debug state
        EDSCR.ITO = '0';        // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO) then
        return;
    EDSCR.ERR = '0';            // Clear cumulative error flag

    return;
```

Library pseudocode for shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =================

bits(2) DebugTargetFrom(boolean secure)
    if HaveEL(EL2) && !secure then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns the status of EDPRSR.EDAD.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if ExternalInvasiveDebugEnabled() then
        if ExternalSecureInvasiveDebugEnabled() then
            return TRUE;
        elsif HaveEL(EL3) then
            return (if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD) == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// ========================
// Returns the status of EDPRSR.EPMAD.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS lock, power-down, etc.
    if ExternalNoninvasiveDebugEnabled() then
        if ExternalSecureNoninvasiveDebugEnabled() then
            return TRUE;
        elsif HaveEL(EL3) then
            return (if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD) == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

Library pseudocode for shared/debug/authentication/Debug_authentication

```
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
```

Library pseudocode for shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================

boolean ExternalInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, ExternalInvasiveDebugEnabled returns the state of the DBGEN
    // signal.
    return DBGEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =================================

boolean ExternalNoninvasiveDebugAllowed()
    // Return TRUE if Trace and PC Sample-based Profiling are allowed
    return (ExternalNoninvasiveDebugEnabled() &&
            (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled() ||
             (ELUsingAArch32(EL1) && PSTATE.EL == EL0 && SDER.SUNIDEN == '1')));
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =================================

boolean ExternalNoninvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN
    // OR NIDEN) signal.
    return ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================

boolean ExternalSecureInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, ExternalSecureInvasiveDebugEnabled returns the state of the
    // (DBGEN AND SPIDEN) signal.
    // CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================

boolean ExternalSecureNoninvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, ExternalSecureNoninvasiveDebugEnabled returns the state of the
    // (DBGEN OR NIDEN) AND (SPIDEN OR SPNIDEN) signal.
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
```

Library pseudocode for shared/debug/cti/CTI_SetEventLevel

```
// Set a Cross Trigger multi-cycle input event trigger to the specified level.
CTI_SetEventLevel(CrossTriggerIn id, signal level);
```

Library pseudocode for shared/debug/cti/CTI_SignalEvent

```
// Signal a discrete event on a Cross Trigger input event trigger.
CTI_SignalEvent(CrossTriggerIn id);
```

Library pseudocode for shared/debug/cti/CrossTrigger

```
enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,  CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,  CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,     CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,         CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,  CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,  CrossTriggerIn_TraceExtOut3};
```

Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    if ELUsingAArch32(EL1) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                   (commtx && DBGDCCINT.TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                   (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRRX_EL0

```
// DBGDTRRX_EL0[] (external write)
// ==============================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xD5330501<31:0>);               // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401<31:0>);               // A64 "STR W1,[X0],#4"
            X[1] = bits(64) UNKNOWN;
        else
            ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
            R[1] = bits(32) UNKNOWN;
        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(32) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
            assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_EL0[] (external read)
// =============================

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRTX_EL0

```
// DBGDTRTX_EL0[] (external read)
// ==============================
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then           // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(32) UNKNOWN;
        else
            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';

        if !UsingAArch32() then
            X[1] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// ===============================

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write
    DTRTX = value;
    return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTR_EL0

```
// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ===================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```

Library pseudocode for shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

Library pseudocode for shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if !Halted() then return;                           // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';              // Overrun condition: block write
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```

Library pseudocode for shared/debug/halting/DCPSInstruction

```
// 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 HaveEL(EL2) && !IsSecure() && HCR_EL2.TGE == '1' then UndefinedFault();
            else handle_el = EL1;

        when EL2
            if !HaveEL(EL2) then UndefinedFault();
            elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
            elsif IsSecure() then UndefinedFault();
            else handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then UndefinedFault();
            handle_el = EL3;
        otherwise
            Unreachable();

    from_secure = IsSecure();
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32();                  // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1
                AArch32.WriteMode(M32_Svc);
                if HavePANExt() && SCTLR.SPAN == '0' then
                    PSTATE.PAN = '1';
            when EL2  AArch32.WriteMode(M32_Hyp);
            when EL3
                AArch32.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();
        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(32) UNKNOWN;  ESR[] = bits(32) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(32) UNKNOWN;

        if HaveUAOExt() then PSTATE.UAO = '0';

    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    // SCTLR[].IESB might be ignored in Debug state.
    if HaveRASExt() && SCTLR[].IESB == '1' && ConstrainUnpredictableBool(Unpredictable_IESBinDebug) the
        ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);
    return;
```

Library pseudocode for shared/debug/halting/DRPSInstruction

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    // SCTLR[].IESB might be ignored in Debug state.
    if HaveRASExt() && SCTLR[].IESB == '1' && ConstrainUnpredictableBool(Unpredictable_IESBinDebug) the
        ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);

    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(32) UNKNOWN;

    UpdateEDSCRFields();                                // Update EDSCR PE state flags.

    return;
```

Library pseudocode for shared/debug/halting/DebugHalt

```
constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```
DisableITRAndResumeInstructionPrefetch();
```

Library pseudocode for shared/debug/halting/ExecuteA64

```
// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);
```

Library pseudocode for shared/debug/halting/ExecuteT32

```
// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);
```

Library pseudocode for shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// ================

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001';                          // Signal restarting
    EDESR<2:0> = '000';                               // Clear any pending Halting debug events

    bits(64) new_pc;
    bits(32) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR);
        spsr = DSPSR;
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr);                           // Can update privileged bits, even at EL0.

    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
        BranchTo(new_pc<31:0>, BranchType_UNKNOWN);    // AArch32 branch
    else
        // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
        if spsr<4> == '1' && ConstrainUnpredictableBool(Unpredictable_RESTARTZEROUPPERPC) then
            new_pc<63:32> = Zeros();
        BranchTo(new_pc, BranchType_DBGEXIT);         // A type of branch that is never predicted

    (EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');       // Atomically signal restarted
    UpdateEDSCRFields();                              // Stop signalling PE state.
    DisableITRAndResumeInstructionPrefetch();

    return;
```

Library pseudocode for shared/debug/halting/Halt

```
// Halt()
// ======

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt
    if UsingAArch32() then
        DLR = ThisInstrAddr();
        DSPSR = GetPSRFromPSTATE();
        DSPSR.SS = PSTATE.SS;                   // Always save PSTATE.SS
    else
        DLR_EL0 = ThisInstrAddr();
        DSPSR_EL0 = GetPSRFromPSTATE();
        DSPSR_EL0.SS = PSTATE.SS;               // Always save PSTATE.SS

    EDSCR.ITE = '1';  EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0';                        // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = (if ExternalSecureInvasiveDebugEnabled() then '0' else '1');
    else
        assert EDSCR.SDD == '1';                // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';
    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if
    // UNKNOWN. PSTATE.{N,Z,C,V,Q,GE} are also not observable, but since these are not changed on
    // exception entry, this function also leaves them unchanged. PSTATE.{E,M,nRW,EL,SP} are
    // unchanged. PSTATE.IL is set to 0.
    if UsingAArch32() then
        PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1'; // PSTATE.J is RES0
    else
        PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                      // Signal entered Debug state
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    return;
```

Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

Library pseudocode for shared/debug/halting/Halted

```
// Halted()
// ========

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});                    // Halted
```

Library pseudocode for shared/debug/halting/HaltingAllowed

```
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elsif 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') 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.
    base = if IsSecure() then 0 else 4;
    if HaltingAllowed() then
        if HaveExtendedECDebugEvents() then
            exception_exit = !exception_entry;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + 8>:EDECCR<UInt(PSTATE.EL) + base>;
            case ctrl of
                when '00'  halt = FALSE;
                when '01'  halt = TRUE;
                when '10'  halt = (exception_exit == TRUE);
                when '11'  halt = (exception_entry == TRUE);
        else
            halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');
        if halt then Halt(DebugHalt_ExceptionCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
```

Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if EDECR.OSUCE == '1' && !Halted() then EDESR.OSUC = '1';
```

Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if EDECR.RCE == '1' then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt(DebugHalt_EDBGRQ);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();
```

Library pseudocode for shared/debug/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// ================

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exeception.
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted();             // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();

    if active && reset then                     // Coming out of reset with EDECR.SS set.
        EDESR.SS = '1';
    elsif active && HaltingAllowed() then
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    case target of
        when EL3
            int_dis = (EDSCR.INTdis == '11' && 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_COMMIRQ, InterruptID_CTIIRQ,
                         InterruptID_COMMRX, InterruptID_COMMTX};
```

Library pseudocode for shared/debug/interrupts/SetInterruptRequestLevel

```
// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);
```

Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// ================

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.

    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ns = if IsSecure() then '0' else '1';
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1;
    if HaveEL(EL2) && !IsSecure() 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() && !IsSecure() && !ELUsingAArch32(EL2) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
    return;
```

Library pseudocode for shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// =================

bits(32) EDPCSRlo[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || EDLSR.SLK == '0';        // Software locked: no side-effects

    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDPCSRhi.PC  = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
                EDPCSRhi.EL  = pc_sample.el;
                EDPCSRhi.NS  = pc_sample.ns;
            else
                EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDVIDSR = pc_sample.contextidr_el2;
            else
                EDVIDSR.VMID = (if HaveEL(EL2) && pc_sample.ns == '1' && pc_sample.el IN {EL1,EL0}
                                then pc_sample.vmid else Zeros(16));
                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,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    bits(16) vmid
)

PCSample pc_sample;
```

Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    if !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() then
        if MDSCR_EL1.SS == '1' && PSTATE.SS == '0' then
            AArch64.SoftwareStepException();
```

Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(32) spsr)
    assert Halted() || Restarting() ||  PSTATE.EL != EL0;

    SS_bit = '0';

    if MDSCR_EL1.SS == '1' then
        if Restarting() then
            enabled_at_source = FALSE;
        elsif UsingAArch32() then
            enabled_at_source = AArch32.GenerateDebugExceptions();
        else
            enabled_at_source = AArch64.GenerateDebugExceptions();

        if IllegalExceptionReturn(spsr) then
            dest = PSTATE.EL;
        else
            (valid, dest) = ELFromSPSR(spsr);  assert valid;

        secure = IsSecureBelowEL3() || dest == EL3;

        if ELUsingAArch32(dest) then
            enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
        else
            mask = spsr<9>;
            enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);

        ELd = DebugTargetFrom(secure);
        if !ELUsingAArch32(ELd) && !enabled_at_source && enabled_at_dest then
            SS_bit = spsr<21>;
    return SS_bit;
```

Library pseudocode for shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean SoftwareStep_DidNotStep();
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean SoftwareStep_SteppedEX();
```

Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// ===================
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    if UsingAArch32() then
        cond = AArch32.CurrentCond();
        if PSTATE.T == '0' then             // A32
            syndrome<4> = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else                                // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            //  * CV set to 0 and COND is set to an UNKNOWN value
            //  * CV set to 1 and COND is set to the condition code for the condition that
            //    applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';

    return syndrome;
```

Library pseudocode for shared/exceptions/exceptions/Exception

```
enumeration Exception {Exception_Uncategorized,       // Uncategorized or unknown reason
                       Exception_WFxTrap,             // Trapped WFI or WFE instruction
                       Exception_CP15RTTrap,          // Trapped AArch32 MCR or MRC access to CP15
                       Exception_CP15RRTTrap,         // Trapped AArch32 MCRR or MRRC access to CP15
                       Exception_CP14RTTrap,          // Trapped AArch32 MCR or MRC access to CP14
                       Exception_CP14DTTrap,          // Trapped AArch32 LDC or STC access to CP14
                       Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
                       Exception_FPIDTrap,            // Trapped access to SIMD or FP ID register
                       // Trapped BXJ instruction not supported in ARMv8
                       Exception_PACTrap,             // Trapped invalid PAC use
                       Exception_CP14RRTTrap,         // Trapped MRRC access to CP14 from AArch32
                       Exception_IllegalState,        // Illegal Execution state
                       Exception_SupervisorCall,      // Supervisor Call
                       Exception_HypervisorCall,      // Hypervisor Call
                       Exception_MonitorCall,         // Monitor Call or Trapped SMC instruction
                       Exception_SystemRegisterTrap,  // Trapped MRS or MSR system register access
                       Exception_ERetTrap,            // Trapped invalid ERET use
                       Exception_InstructionAbort,    // Instruction Abort or Prefetch Abort
                       Exception_PCAlignment,         // PC alignment fault
                       Exception_DataAbort,           // Data Abort
                       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_SoftwareBreakpoint,  // Software Breakpoint Instruction
                       Exception_VectorCatch,         // AArch32 Vector Catch
                       Exception_IRQ,                 // IRQ interrupt
                       Exception_FIQ};                // FIQ interrupt
```

Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```
type ExceptionRecord is (Exception type,            // Exception class
                         bits(25) syndrome,         // Syndrome record
                         bits(64) vaddress,         // Virtual fault address
                         boolean ipavalid,          // Physical fault address is valid
                         bits(52) ipaddress)        // Physical fault address for second stage faults
```

Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception type)

    ExceptionRecord r;

    r.type = type;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.ipaddress = Zeros();

    return r;
```

Library pseudocode for shared/exceptions/traps/ReservedValue

```
// ReservedValue()
// ===============

ReservedValue()

    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();
```

Library pseudocode for shared/exceptions/traps/SystemAccessType

```
enumeration SystemAccessType { SystemAccessType_RT,  SystemAccessType_RRT,  SystemAccessType_DT };
```

Library pseudocode for shared/exceptions/traps/UnallocatedEncoding

```
// UnallocatedEncoding()
// =====================

UnallocatedEncoding()
    if UsingAArch32() && AArch32.ExecutingCP10or11Instr() then
        FPEXC.DEX = '0';
    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();
```

Library pseudocode for shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault type, integer level)

    bits(6) result;
    case type of
        when Fault_AddressSize         result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag          result = '0010':level<1:0>; assert level IN {1,2,3};
        when Fault_Permission          result = '0011':level<1:0>; assert level IN {1,2,3};
        when Fault_Translation         result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal        result = '010000';
        when Fault_SyncExternalOnWalk  result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity          result = '011000';
        when Fault_SyncParityOnWalk    result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity         result = '011001';
        when Fault_AsyncExternal       result = '010001';
        when Fault_Alignment           result = '100001';
        when Fault_Debug               result = '100010';
        when Fault_TLBConflict         result = '110000';
        when Fault_Lockdown            result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive           result = '110101';  // IMPLEMENTATION DEFINED
        otherwise                      Unreachable();

    return result;
```

Library pseudocode for shared/functions/aborts/IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.type != Fault_None;

    if fault.s2fs1walk then
        return fault.type IN {Fault_AccessFlag, Fault_Permission, Fault_Translation,
                              Fault_AddressSize};
    elsif fault.secondstage then
        return fault.type IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
    else
        return FALSE;
```

Library pseudocode for shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.type);
```

Library pseudocode for shared/functions/aborts/IsDebugException

```
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.type != Fault_None;
    return fault.type == Fault_Debug;
```

Library pseudocode for shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =================
// Returns TRUE if the abort currently being processed is an external abort and FALSE otherwise.

boolean IsExternalAbort(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOn
                     Fault_AsyncExternal, Fault_AsyncParity });

// IsExternalAbort()
// =================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.type);
```

Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE other

boolean IsExternalSyncAbort(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOn

// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.type);
```

Library pseudocode for shared/functions/aborts/IsFault

```
// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.type != Fault_None;
```

Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault type)
    assert type != Fault_None;

    return (type IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.type);
```

Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.type != Fault_None;

    return fault.secondstage;
```

Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

```
bits(11) LSInstructionSyndrome();
```

Library pseudocode for shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/ASR_C

```
// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    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 - 1 - HighestSetBit(x);
```

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/NOT

```
bits(N) NOT(bits(N) x);
```

Library pseudocode for shared/functions/common/Ones

```
// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// ======

bits(N) Ones()
    return Ones(N);
```

Library pseudocode for shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/ROR_C

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/Replicate

```
// Replicate()
// ===========

bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);
```

Library pseudocode for shared/functions/common/RoundDown

```
integer RoundDown(real x);
```

Library pseudocode for shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

Library pseudocode for shared/functions/common/RoundUp

```
integer RoundUp(real x);
```

Library pseudocode for shared/functions/common/SInt

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

Library pseudocode for shared/functions/common/SignExtend

```
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;

// SignExtend()
// ============

bits(N) SignExtend(bits(M) x)
    return SignExtend(x, N);
```

Library pseudocode for shared/functions/common/UInt

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

Library pseudocode for shared/functions/common/ZeroExtend

```
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;

// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =======

bits(N) Zeros(integer N)
    return Replicate('0',N);

// Zeros()
// =======

bits(N) Zeros()
    return Zeros(N);
```

Library pseudocode for shared/functions/crc/BitReverse

```
// BitReverse()
// ============

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<N-i-1> = data<i>;
    return result;
```

Library pseudocode for shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// ============

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

Library pseudocode for shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
    assert N > 32;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR poly:Zeros(i-32);
    return data<31:0>;
```

Library pseudocode for shared/functions/crypto/AESInvMixColumns

```
bits(128) AESInvMixColumns(bits (128) op);
```

Library pseudocode for shared/functions/crypto/AESInvShiftRows

```
bits(128) AESInvShiftRows(bits(128) op);
```

Library pseudocode for shared/functions/crypto/AESInvSubBytes

```
bits(128) AESInvSubBytes(bits(128) op);
```

Library pseudocode for shared/functions/crypto/AESMixColumns

```
bits(128) AESMixColumns(bits (128) op);
```

Library pseudocode for shared/functions/crypto/AESShiftRows

```
bits(128) AESShiftRows(bits(128) op);
```

Library pseudocode for shared/functions/crypto/AESSubBytes

```
bits(128) AESSubBytes(bits(128) op);
```

Library pseudocode for shared/functions/crypto/HaveChCryptoExt

```
// HaveChCryptoExt()
// =================
// TRUE if SM3 and SM4 extended cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveChCryptoExt()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM3 and SM4 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveCryptoExt

```
// HaveCryptoExt()
// ===============
// TRUE if AES, SHA1, SHA2 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveCryptoExt()
    return boolean IMPLEMENTATION_DEFINED "Has basic Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveCryptoExt2

```
// HaveCryptoExt2()
// ================
// TRUE if SHA512 and SHA3 extended cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveCryptoExt2()
    if !HasArchVersion(ARMv8p2) || !HaveCryptoExt() then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 and SHA3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

Library pseudocode for shared/functions/crypto/SHA256hash

```
// SHA256hash()
// ============

bits(128) SHA256hash(bits (128) X, bits(128) Y, bits(128) W, boolean part1)
    bits(32) chs, maj, t;

    for e = 0 to 3
        chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
        maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
        t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
        X<127:96> = t + X<127:96>;
        Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
        <Y, X> = ROL(Y : X, 32);
    return (if part1 then X else Y);
```

Library pseudocode for shared/functions/crypto/SHAchoose

```
// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// ===============

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// ===============

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

Library pseudocode for shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =============

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));
```

Library pseudocode for shared/functions/crypto/SHAparity

```
// SHAparity()
// ===========

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

Library pseudocode for shared/functions/crypto/Sbox

```
// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = 0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f49

    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// ========================

// Clear the local Exclusives monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

```
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();
```

Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

```
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

```
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// ====================

boolean AArch32.HaveHPDExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// ====================

boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// ================

boolean Have52BitPAExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// ================

boolean Have52BitVAExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// ===============

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// ==============================

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()
// =============
// Returns TRUE if has Dot Product feature support, and FALSE otherwise.

boolean HaveDOTPExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has Dot Product extension";
```

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/HaveFJCVTZSExt

```
// HaveFJCVTZSExt()
// ================

boolean HaveFJCVTZSExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// =============

boolean HaveHPMDExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveNVExt

```
// HaveNVExt()
// ===========

boolean HaveNVExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HavePANExt

```
// HavePANExt()
// ============

boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()
// =================================

boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HavePrivATExt

```
// HavePrivATExt()
// ===============

boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt()
// ================

boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveRASExt

```
// HaveRASExt()
// ============

boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) ||
            boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfiling

```
// HaveStatisticalProfiling()
// ==========================

boolean HaveStatisticalProfiling()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```
// HaveTrapLoadStoreMultipleDeviceExt()
// ====================================

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveUAOExt

```
// HaveUAOExt()
// ============

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt()
// =================

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/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};
    return '0' : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

Library pseudocode for shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = '0011';
        if type1==FPType_SNaN || type2==FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';
    return result;
```

Library pseudocode for shared/functions/float/fpcompareeq/FPCompareEQ

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        if type1==FPType_SNaN || type2==FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;
```

Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

```
// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
    return result;
```

Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
    return result;
```

Library pseudocode for shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding)
    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (type,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if type == FPType_SNaN || type == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        if type == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif type == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCV(value, fpcr, rounding);
    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType 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/FPCRType

```
type FPCRType;
```

Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
    case rm of
        when '00' return FPRounding_TIEAWAY; // A
        when '01' return FPRounding_TIEEVEN; // N
        when '10' return FPRounding_POSINF;  // P
        when '11' return FPRounding_NEGINF;  // M
```

Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// ==================

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF;  // P
        when '10' return FPRounding_NEGINF;  // M
        when '11' return FPRounding_ZERO;    // Z
```

Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN()
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = '0';
    exp  = Ones(E);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpexc/FPExc

```
enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                         FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Ones(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 > value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType_Infinity then
            result = FPInfinity(sign);
        elsif type == FPType_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign);
        else
            result = FPRound(value, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // treat a single quiet-NaN as -Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1');

    return FPMax(op1, op2, fpcr);
```

Library pseudocode for shared/functions/float/fpmin/FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 < value2 then
            (type,sign,value) = (type1,sign1,value1);
        else
            (type,sign,value) = (type2,sign2,value2);
        if type == FPType_Infinity then
            result = FPInfinity(sign);
        elsif type == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            result = FPRound(value, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // Treat a single quiet-NaN as +Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('0');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('0');

    return FPMin(op1, op2, fpcr);
```

Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd

```
// FPMulAdd()
// ==========
//
// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);

    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity);  zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH

```
// FPMulAddH()
// ===========

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);
    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;
        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');
        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);
        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3, bits(N) op1, bits(N DIV 2)
    assert N IN {32,64};
    bits(N) result;
    if type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
    else
        done = FALSE; result = Zeros(); // 'Don't care' result
    return (done, result);
```

Library pseudocode for shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)
    assert N IN {16,32,64};
    return NOT(op<N-1>) : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)
    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp      cumul = 0;
        when FPExc_DivideByZero   cumul = 1;
        when FPExc_Overflow       cumul = 2;
        when FPExc_Underflow      cumul = 3;
        when FPExc_Inexact        cumul = 4;
        when FPExc_InputDenorm    cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrapException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType type, bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    assert type IN {FPType_QNaN, FPType_SNaN};

    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if type == FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN();
    return result;
```

Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                 bits(N) op1, bits(N) op2,
                                 FPCRType fpcr)
    assert N IN {16,32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

Library pseudocode for shared/functions/float/fpprocessnans3/FPProcessNaNs3

```
// FPProcessNaNs3()
// ================
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRType fpcr)
    assert N IN {16,32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

Library pseudocode for shared/functions/float/fprecipestimate/FPRecipEstimate

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPType_Infinity then
        result = FPZero(sign);
    elsif type == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (
            (N == 16 && Abs(value) < 2.0^-16) ||
            (N == 32 && Abs(value) < 2.0^-128) ||
            (N == 64 && Abs(value) < 2.0^-1024)
          ) then
        case FPRoundingMode(fpcr) of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
          && (
                (N == 16 && Abs(value) >= 2.0^14) ||
                (N == 32 && Abs(value) >= 2.0^126) ||
                (N == 64 && Abs(value) >= 2.0^1022)
             ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign);
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            if fraction<51> == 0 then
                exp = -1;
                fraction = fraction<49:0>:'00';
            else
                fraction = fraction<50:0>:'0';

        integer scaled = UInt('1':fraction<51:44>);

        case N of
            when 16 result_exp =   29 - exp; // In range 29-30 = -1 to 29+1 = 30
            when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
```

```
            when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

        // scaled is in range 256..511 representing a fixed-point number in range [0.5..1.0)
        estimate = RecipEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
        // Convert to scaled floating point result with copied sign bit,
        // high-order bits from estimate, and exponent calculated above.

        fraction = estimate<7:0> : Zeros(44);
        if result_exp == 0 then
            fraction = '1' : fraction<51:1>;
        elsif result_exp == -1 then
            fraction = '01' : fraction<51:2>;
            result_exp = 0;

        case N of
            when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
            when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
            when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

    return result;
```

Library pseudocode for shared/functions/float/fprecipestimate/RecipEstimate

```
// Compute estimate of reciprocal of 9-bit fixed-point number
//
// a is in range 256 .. 511 representing a number in the range 0.5 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipEstimate(integer a)
    assert 256 <= a && a < 512;
    a = a*2+1; // round to nearest
    integer b = (2 ^ 19) DIV a;
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

Library pseudocode for shared/functions/float/fprecpx/FPRecpX

```
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};

    case N of
        when 16 esize =  5;
        when 32 esize =  8;
        when 64 esize = 11;

    bits(N)          result;
    bits(esize)      exp;
    bits(esize)      max_exp;
    bits(N-esize-1)  frac = Zeros();

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp = Ones(esize) - 1;

    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    else
        if IsZero(exp) then // Zero and denormals
            result = sign:max_exp:frac;
        else // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;
```

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, FPCRType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    return FPRoundBase(op, fpcr, rounding);

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Deal with flush-to-zero.
    if ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) && exponent < minimum_exp then
        // Flush-to-zero never generates a trapped exception
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            FPSR.UFC = '1';
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped.
    if biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
        FPProcessException(FPExc_Underflow, fpcr);

    // Round result according to rounding mode.
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up = (error != 0.0 && sign == '1');
```

```
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then      // Rounded up from denormalized to normalized
            biased_exp = 1;
        if int_mant == 2^(F+1) then  // Rounded up to next exponent
            biased_exp = biased_exp + 1;  int_mant = int_mant DIV 2;

    // Handle rounding to odd aka Von Neumann rounding
    if error != 0.0 && rounding == FPRounding_ODD then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
            FPProcessException(FPExc_Overflow, fpcr);
            error = 1.0;  // Ensure that an Inexact exception occurs
        else
            result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;
    else                                     // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0;  // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<N-F-2:0> : int_mant<F-1:0>;

    // Deal with Inexact exception.
    if error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
```

Library pseudocode for shared/functions/float/fpround/FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr, FPRounding rounding)
    fpcr.FZ16 = '0';
    return FPRoundBase(op, fpcr, rounding);
```

Library pseudocode for shared/functions/float/fprounding/FPRounding

```
enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
                         FPRounding_NEGINF,  FPRounding_ZERO,
                         FPRounding_TIEAWAY, FPRounding_ODD};
```

Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

```
// FPRoundingMode()
// ================

// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
    return FPDecodeRounding(fpcr.RMode);
```

Library pseudocode for shared/functions/float/fproundint/FPRoundInt

```
// FPRoundInt()
// ============

// Round OP to nearest integral floating point value using rounding mode ROUNDING.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to OP.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)
    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPType_Infinity then
        result = FPInfinity(sign);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    else
        // extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact
        if real_result == 0.0 then
            result = FPZero(sign);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO);

        // Generate inexact exceptions
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(operand, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, fpcr);
    elsif type == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif type == FPType_Infinity then
        result = FPZero('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == 0 do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('01':fraction<51:45>);

        case N of
            when 16 result_exp = (  44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros( 2);
            when 32 result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
            when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);
    return result;
```

Library pseudocode for shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

```
// Compute estimate of reciprocal square root of 9-bit fixed-point number
//
// a is in range 128 .. 511 representing a number in the range 0.25 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipSqrtEstimate(integer a)
    assert 128 <= a && a < 512;
    if a < 256 then // 0.25 .. 0.5
        a = a*2+1;     // a in units of 1/512 rounded to nearest
    else // 0.5 .. 1.0
        a = (a >> 1) << 1;   // discard bottom bit
        a = (a+1)*2;  // a in units of 1/256 rounded to nearest
    integer b = 512;
    while a*(b+1)*(b+1) < 2^28 do
        b = b+1;
    // b = largest b such that b < 2^14 / sqrt(a) do
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    (type,sign,value) = FPUnpack(op, fpcr);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, op, fpcr);
    elsif type == FPType_Zero then
        result = FPZero(sign);
    elsif type == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr);
    return result;
```

Library pseudocode for shared/functions/float/fpsub/FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

Library pseudocode for shared/functions/float/fpthree/FPThree

```
// FPThree()
// =========

bits(N) FPThree(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    // If NaN, set cumulative flag or take exception
    if type == FPType_SNaN || type == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

```
// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

bits(N) FPToFixedJS(bits(M) op, FPCRType fpcr, boolean Is64)

    assert M == 64 && N == 32;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (type,sign,value) = FPUnpack(op, fpcr);

    Z = '1';
    // If NaN, set cumulative flag or take exception
    if type == FPType_SNaN || type == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        Z = '0';
    if sign == '1'&& value == 0.0 then
        Z = '0';

    if type == FPType_Infinity then result = 0;

    if Is64 then
        PSTATE.<N,Z,C,V> = '0':Z:'00';
    else
        FPSCR<31:28> = '0':Z:'00';

    return result<31:0>;
```

Library pseudocode for shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =======

bits(N) FPTwo(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fptype/FPType

```
enumeration FPType      {FPType_Nonzero, FPType_Zero, FPType_Infinity,
                         FPType_QNaN, FPType_SNaN};
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// ==========
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpackBase

```
// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr)
    assert N IN {16,32,64};

    if N == 16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            // Produce zero if value is zero or flush-to-zero is selected
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                type = FPType_Zero;  value = 0.0;
            else
                type = FPType_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 then

        sign   = fpval<31>;
        exp32  = fpval<30:23>;
        frac32 = fpval<22:0>;
        if IsZero(exp32) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac32) || fpcr.FZ == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac32) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpcr);
            else
                type = FPType_Nonzero;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64

        sign   = fpval<63>;
        exp64  = fpval<62:52>;
        frac64 = fpval<51:0>;
        if IsZero(exp64) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac64) || fpcr.FZ == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac64) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpcr);
            else
```

```
                type = FPType_Nonzero;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                type = FPType_Infinity;  value = 2.0^1000000;
            else
                type = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;
    return (type, sign, value);
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr)
    fpcr.FZ16 = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

Library pseudocode for shared/functions/float/fpzero/FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    exp  = Zeros(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/vfpexpandimm/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);
    return sign : exp : frac;
```

Library pseudocode for shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

Library pseudocode for shared/functions/memory/AArch64.BranchAddr

```
// AArch64.BranchAddr()
// ====================
// Return the virtual address with tag bits removed for storing to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, PSTATE.EL);
    if msbit == 63 then
        return vaddress;
    elsif (PSTATE.EL IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>);
    else
        return ZeroExtend(vaddress<msbit:0>);
```

Library pseudocode for shared/functions/memory/AccType

```
enumeration AccType {AccType_NORMAL, AccType_VEC,         // Normal loads and stores
                     AccType_STREAM, AccType_VECSTREAM,   // Streaming loads and stores
                     AccType_ATOMIC, AccType_ATOMICRW,    // Atomic loads and stores
                     AccType_ORDERED, AccType_ORDEREDRW,  // Load-Acquire and Store-Release
                     AccType_LIMITEDORDERED,              // Load-LOAcquire and Store-LORelease
                     AccType_UNPRIV,                      // Load and store unprivileged
                     AccType_IFETCH,                      // Instruction fetch
                     AccType_PTW,                         // Page table walk
                     // Other operations
                     AccType_DC,                          // Data cache maintenance
                     AccType_IC,                          // Instruction cache maintenance
                     AccType_DCZVA,                       // DC ZVA instructions
                     AccType_AT};                         // Address translation
```

Library pseudocode for shared/functions/memory/AccessDescriptor

```
type AccessDescriptor is (
    AccType acctype,
    boolean page_table_walk,
    boolean secondstage,
    boolean s2fs1walk,
    integer level
)
```

Library pseudocode for shared/functions/memory/AddrTop

```
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        // AArch64 translation regime.
        case regime of
            when EL1
                tbi = (if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0);
                if HavePACExt() then
                    tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
            when EL2
                if HaveVirtHostExt() && ELIsInHost(el) then
                    tbi = (if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0);
                    if HavePACExt() then
                        tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
                else
                    tbi = TCR_EL2.TBI;
                    if HavePACExt() then tbid = TCR_EL2.TBID;
            when EL3
                tbi = TCR_EL3.TBI;
                if HavePACExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr ) then 55 else 63);
```

Library pseudocode for shared/functions/memory/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord      fault,      // fault.type indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    bits(64)         vaddress
)
```

Library pseudocode for shared/functions/memory/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()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

Library pseudocode for shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// ==================

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```

Library pseudocode for shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```

Library pseudocode for shared/functions/memory/CreateAccessDescriptor

```
// CreateAccessDescriptor()
// ========================

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.page_table_walk = FALSE;
    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccessDescriptorPTW

```
// CreateAccessDescriptorPTW()
// ===========================

AccessDescriptor CreateAccessDescriptorPTW(AccType acctype, boolean secondstage,
                                           boolean s2fs1walk, integer level)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.page_table_walk = TRUE;
    accdesc.secondstage = s2fs1walk;
    accdesc.secondstage = secondstage;
    accdesc.level = level;
    return accdesc;
```

Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/memory/DescriptorUpdate

```
type DescriptorUpdate is (
    boolean AF,                 // AF needs to be set
    boolean AP,                 // AP[2] / S2AP[2] will be modified
    AddressDescriptor descaddr  // Descriptor to be updated
)
```

Library pseudocode for shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

Library pseudocode for shared/functions/memory/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_Lockdown,
                   Fault_Exclusive,
                   Fault_ICacheMaint};
```

Library pseudocode for shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault    type,          // Fault Status
                     AccType  acctype,       // Type of access that faulted
                     bits(52) ipaddress,     // Intermediate physical address
                     boolean  s2fs1walk,     // Is on a Stage 1 page table walk
                     boolean  write,         // TRUE for a write, FALSE for a read
                     integer  level,         // For translation, access flag and permission faults
                     bit      extflag,       // IMPLEMENTATION DEFINED syndrome for external aborts
                     boolean  secondstage,   // Is a Stage 2 abort
                     bits(4)  domain,        // Domain number, AArch32 only
                     bits(2)  errortype,      // [ARMv8.2 RAS] AArch32 AET or AArch64 SET
                     bits(4)  debugmoe)      // Debug method of entry, from AArch32 only
```

Library pseudocode for shared/functions/memory/FullAddress

```
type FullAddress is (
    bits(52) physicaladdress,
    bit      NS                   // '0' = Secure, '1' = Non-secure
)
```

Library pseudocode for shared/functions/memory/Hint_Prefetch

```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visiblestate should be on caches and TLBs associated with address, which must be
// accessable by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

Library pseudocode for shared/functions/memory/MBReqDomain

```
enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

Library pseudocode for shared/functions/memory/MBReqTypes

```
enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

Library pseudocode for shared/functions/memory/MemAttrHints

```
type MemAttrHints is (
    bits(2) attrs,  // The possible encodings for each attributes field are as below
    bits(2) hints,  // The possible encodings for the hints are below
    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      type,

    DeviceType   device,     // For Device memory types
    MemAttrHints inner,      // Inner hints and attributes
    MemAttrHints outer,      // Outer hints and attributes

    boolean      shareable,
    boolean      outershareable
)
```

Library pseudocode for shared/functions/memory/Permissions

```
type Permissions is (
    bits(3) ap,   // Access permission bits
    bit     xn,   // Execute-never bit
    bit     xxn,  // [ARMv8.2] Extended execute-never bit for stage 2
    bit     pxn   // Privileged execute-never bit
)
```

Library pseudocode for shared/functions/memory/PrefetchHint

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

Library pseudocode for shared/functions/memory/TLBRecord

```
type TLBRecord is (
    Permissions       perms,
    bit               nG,             // '0' = Global, '1' = not Global
    bits(4)           domain,         // AArch32 only
    boolean           contiguous,     // Contiguous bit from page table
    integer           level,          // AArch32 Short-descriptort format: Indicates Section/Page
    integer           blocksize,      // Describes size of memory translated in KBytes
    DescriptorUpdate  descupdate,     // [ARMv8.1] Context for h/w update of table descriptor
    bit               CnP,            // [ARMv8.2] TLB entry can be shared between different PEs
    AddressDescriptor addrdesc
)
```

Library pseudocode for shared/functions/memory/_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/registers/BranchTo

```
// BranchTo()
// ==========

// Set program counter to a new address, which might include a tag in the top eight bits,
// with a branch reason hint for possible use by hardware fetching the next instruction.

BranchTo(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = AArch64.BranchAddr(target<63:0>);
    return;
```

Library pseudocode for shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// ==============

// Set program counter to a new address, which does not include a tag in the top eight bits,
// with a branch reason hint for possible use by hardware fetching the next instruction.

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_CALL, BranchType_ERET, BranchType_DBGEXIT,
                        BranchType_RET, BranchType_JMP, BranchType_EXCEPTION,
                        BranchType_UNKNOWN};
```

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 next instruction.
bits(N) NextInstrAddr();
```

Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

```
// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);
```

Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr()
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

Library pseudocode for shared/functions/registers/_PC

```
bits(64) _PC;
```

Library pseudocode for shared/functions/registers/_R

```
array bits(64) _R[0..30];
```

Library pseudocode for shared/functions/registers/_V

```
array bits(128) _V[0..31];
```

Library pseudocode for shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// ============================

bits(32) SPSR[]
    bits(32) result;
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ       result = SPSR_fiq;
            when M32_IRQ       result = SPSR_irq;
            when M32_Svc       result = SPSR_svc;
            when M32_Monitor   result = SPSR_mon;
            when M32_Abort     result = SPSR_abt;
            when M32_Hyp       result = SPSR_hyp;
            when M32_Undef     result = SPSR_und;
            otherwise          Unreachable();
    else
        case PSTATE.EL of
            when EL1           result = SPSR_EL1;
            when EL2           result = SPSR_EL2;
            when EL3           result = SPSR_EL3;
            otherwise          Unreachable();

    return result;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(32) value
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ       SPSR_fiq = value;
            when M32_IRQ       SPSR_irq = value;
            when M32_Svc       SPSR_svc = value;
            when M32_Monitor   SPSR_mon = value;
            when M32_Abort     SPSR_abt = value;
            when M32_Hyp       SPSR_hyp = value;
            when M32_Undef     SPSR_und = value;
            otherwise          Unreachable();
    else
        case PSTATE.EL of
            when EL1           SPSR_EL1 = value;
            when EL2           SPSR_EL2 = value;
            when EL3           SPSR_EL3 = value;
            otherwise          Unreachable();

    return;
```

Library pseudocode for shared/functions/system/ArchVersion

```
enumeration ArchVersion {
    ARMv8p0
    , ARMv8p1
    , ARMv8p2
    , ARMv8p3
};
```

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/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/CurrentInstrSet

```
// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()

    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;
```

Library pseudocode for shared/functions/system/CurrentPL

```
// CurrentPL()
// ===========

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);
```

Library pseudocode for shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

Library pseudocode for shared/functions/system/ELFromM32

```
// ELFromM32()
// ===========

(boolean,bits(2)) ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid,EL):
    //   'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    //           and the current value of SCR.NS/SCR_EL3.NS.
    //   'EL'    is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = !BadMode(mode);  // Check for modes that are not valid for this implementation
    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
            valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && HighestELUsingAArch32() && SCR.NS == '0' then EL3 else EL1);
        when M32_User
            el = EL0;
        otherwise
            valid = FALSE;           // Passed an illegal mode value
    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

Library pseudocode for shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(32) spsr)
    if spsr<4> == '0' then                              // AArch64 state
        el = spsr<3:2>;
        if HighestELUsingAArch32() then                 // No AArch64 support
            valid = FALSE;
        elsif !HaveEL(el) then                          // Exception level not implemented
            valid = FALSE;
        elsif spsr<1> == '1' then                       // M[1] must be 0
            valid = FALSE;
        elsif el == EL0 && spsr<0> == '1' then          // for EL0, M[0] must be 0
            valid = FALSE;
        elsif el == EL2 && HaveEL(EL3) && SCR_EL3.NS == '0' then
            valid = FALSE;                              // EL2 only valid in Non-secure state
        else
            valid = TRUE;
    elsif !HaveAnyAArch32() then                        // AArch32 not supported
        valid = FALSE;
    else                                                // AArch32 state
        (valid, el) = ELFromM32(spsr<4:0>);
    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
```

Library pseudocode for shared/functions/system/ELIsInHost

```
// ELIsInHost()
// ============
// Returns TRUE if HaveVirtHostExt() is TRUE and executing at 'el' would mean be executing within
// an EL2 Host OS or an EL0 application of a Host OS in Non-secure state using AArch64 with
// HCR_EL2.E2H set to 1, and FALSE otherwise.

boolean ELIsInHost(bits(2) el)
    return (!IsSecureBelowEL3() && HaveVirtHostExt() && !ELUsingAArch32(EL2) &&
            HCR_EL2.E2H == '1' && (el == EL2 || (el == EL0 && HCR_EL2.TGE == '1')));
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====================

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// ======================

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    // Returns (known, aarch32):
    //   'known'   is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
    //             using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
    //   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
    boolean aarch32;
    known = TRUE;
    if !HaveAArch32EL(el) then
        aarch32 = FALSE;                          // Exception level is using AArch64
    elsif HighestELUsingAArch32() then
        aarch32 = TRUE;                           // All levels are using AArch32
    else
        aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0';

        aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) && !secure && HCR_EL2.RW == '0' &&
                          !(HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt())));
        if el == EL0 && !aarch32_at_el1 then      // Only know if EL0 using AArch32 from PSTATE
            if PSTATE.EL == EL0 then
                aarch32 = PSTATE.nRW == '1';      // EL0 controlled by PSTATE
            else
                known = FALSE;                    // EL0 state is UNKNOWN
        else
            aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});
    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);
```

Library pseudocode for shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/EndOfInstruction

```
// Terminate processing of the current instruction.
EndOfInstruction();
```

Library pseudocode for shared/functions/system/EnterLowPowerState

```
// PE enters a low-power state
EnterLowPowerState();
```

Library pseudocode for shared/functions/system/ErrorSynchronizationBarrier

```
// Perform an Error Synchronization Barrier operation
ErrorSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(32) GetPSRFromPSTATE()
    bits(32) spsr = Zeros();
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<21>    = PSTATE.SS;
    spsr<20>    = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        spsr<27>    = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if HaveUAOExt() then spsr<23> = '0';
        spsr<19:16> = PSTATE.GE;
        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9>     = PSTATE.E;
        spsr<8:6>   = PSTATE.<A,I,F>;                   // No PSTATE.D in AArch32 state
        spsr<5>     = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW;               // bit [4] is the discriminator
        spsr<4:0>   = PSTATE.M;
    else // AArch64 state
        if HaveUAOExt() then spsr<23> = PSTATE.UAO;
        spsr<9:6>   = PSTATE.<D,A,I,F>;
        spsr<4>     = PSTATE.nRW;
        spsr<3:2>   = PSTATE.EL;
        spsr<0>     = PSTATE.SP;
    return spsr;
```

Library pseudocode for shared/functions/system/HasArchVersion

```
// HasArchVersion()
// ================

// Return TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// ===============

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE;                    // The Exception level is not implemented
    elsif !HaveAnyAArch32() then
        return FALSE;                    // No Exception level can use AArch32
    elsif HighestELUsingAArch32() then
        return TRUE;                     // All Exception levels are using AArch32
    elsif el == HighestEL() then
        return FALSE;                    // The highest Exception level is using AArch64
    elsif el == EL0 then
        return TRUE;                     // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAnyAArch32

```
// HaveAnyAArch32()
// ================
// Return TRUE if AArch32 state is supported at any Exception level

boolean HaveAnyAArch32()
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAnyAArch64

```
// HaveAnyAArch64()
// ================
// Return TRUE if AArch64 state is supported at any Exception level

boolean HaveAnyAArch64()
    return !HighestELUsingAArch32();
```

Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    if el IN {EL1,EL0} then
        return TRUE;                             // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =============
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HighestEL

```
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

Library pseudocode for shared/functions/system/HighestELUsingAArch32

```
// HighestELUsingAArch32()
// =======================
// Return TRUE if configured to boot into AArch32 operation

boolean HighestELUsingAArch32()
    if !HaveAnyAArch32() then return FALSE;
    return boolean IMPLEMENTATION_DEFINED;       // e.g. CFG32SIGNAL == HIGH
```

Library pseudocode for shared/functions/system/Hint_Yield

```
Hint_Yield();
```

Library pseudocode for shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(32) spsr)

    // Check for return:
    //   * To an unimplemented Exeception level.
    //   * To EL2 in Secure state.
    //   * 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 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 return to EL1 in Non-secure state when HCR.TGE is set
    if HaveEL(EL2) && target == EL1 && !IsSecureBelowEL3() && HCR_EL2.TGE == '1' 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

```
// Return TRUE if there are any pending physical or virtual interrupts, and FALSE otherwise
boolean InterruptPending();
```

Library pseudocode for shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE otherwise

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

Library pseudocode for shared/functions/system/IsInHost

```
// IsInHost()
// ==========
// Returns TRUE if HaveVirtHostExt() is TRUE and executing within a Host OS or an EL0 application
// of a Host OS using AArch64 with HCR_EL2.E2H set to 1, and FALSE otherwise.

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

Library pseudocode for shared/functions/system/IsSecure

```
// IsSecure()
// ==========

boolean IsSecure()
    // Return TRUE if current Exception level is in Secure state.
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elsif 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';
    elsif HaveEL(EL2) then
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure;
        return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/Mode_Bits

```
constant bits(5) M32_User    = '10000';
constant bits(5) M32_FIQ     = '10001';
constant bits(5) M32_IRQ     = '10010';
constant bits(5) M32_Svc     = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort   = '10111';
constant bits(5) M32_Hyp     = '11010';
constant bits(5) M32_Undef   = '11011';
constant bits(5) M32_System  = '11111';
```

Library pseudocode for shared/functions/system/PLOfEL

```
// PLOfEL()
// ========

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3  return if HighestELUsingAArch32() then PL1 else PL3;
        when EL2  return PL2;
        when EL1  return PL1;
        when EL0  return PL0;
```

Library pseudocode for shared/functions/system/PSTATE

```
ProcState PSTATE;
```

Library pseudocode for shared/functions/system/PrivilegeLevel

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

Library pseudocode for shared/functions/system/ProcState

```
type ProcState is (
    bits (1) N,        // Negative condition flag
    bits (1) Z,        // Zero condition flag
    bits (1) C,        // Carry condition flag
    bits (1) V,        // oVerflow condition flag
    bits (1) D,        // Debug mask bit                     [AArch64 only]
    bits (1) A,        // SError interrupt mask bit
    bits (1) I,        // IRQ mask bit
    bits (1) F,        // FIQ mask bit
    bits (1) PAN,      // Privileged Access Never Bit        [v8.1]
    bits (1) UAO,      // User Access Override               [v8.2]
    bits (1) 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 (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(32) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

Library pseudocode for shared/functions/system/SCRType

```
type SCRType;
```

Library pseudocode for shared/functions/system/SCR_GEN

```
// SCR_GEN[]
// =========

SCRType SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(32) r;
    if HighestELUsingAArch32() then
        r = SCR;
    else
        r = SCR_EL3;
    return r;
```

Library pseudocode for shared/functions/system/SErrorPending

```
// Return TRUE if a physical SError interrupt is pending; that is, if ISR_EL1.A == 1
boolean SErrorPending();
```

Library pseudocode for shared/functions/system/SendEvent

```
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed
SendEvent();
```

Library pseudocode for shared/functions/system/SendEventLocal

```
// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed

SendEventLocal()
    EventRegister = '1';
    return;
```

Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(32) spsr)

    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then                    // AArch32 state
            AArch32.WriteMode(spsr<4:0>);         // Sets PSTATE.EL correctly
        else                                      // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL  = spsr<3:2>;
            PSTATE.SP  = spsr<0>;
    // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
    // the T bit is set to zero or copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';

    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if PSTATE.nRW == '1' then                     // AArch32 state
        PSTATE.Q         = spsr<27>;
        PSTATE.IT        = RestoredITBits(spsr);
        PSTATE.GE        = spsr<19:16>;
        PSTATE.E         = spsr<9>;
        PSTATE.<A,I,F>   = spsr<8:6>;             // No PSTATE.D in AArch32 state
        PSTATE.T         = spsr<5>;               // PSTATE.J is RES0
    else                                          // AArch64 state
        PSTATE.<D,A,I,F> = spsr<9:6>;             // No PSTATE.<Q,IT,GE,E,T> in AArch64 state

    if HavePANExt() then PSTATE.<PAN> = spsr<22>;
    if HaveUAOExt() then PSTATE.UAO = spsr<23>;
    return;
```

Library pseudocode for shared/functions/system/SynchronizeContext

```
SynchronizeContext();
```

Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.
TakeUnmaskedSErrorInterrupts();
```

Library pseudocode for shared/functions/system/ThisInstr

```
bits(32) ThisInstr();
```

Library pseudocode for shared/functions/system/ThisInstrLength

```
integer ThisInstrLength();
```

Library pseudocode for shared/functions/system/Unreachable

```
Unreachable()
    assert FALSE;
```

Library pseudocode for shared/functions/system/UsingAArch32

```
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAnyAArch32() then assert !aarch32;
    if HighestELUsingAArch32() then assert aarch32;
    return aarch32;
```

Library pseudocode for shared/functions/system/WaitForEvent

```
// WaitForEvent()
// ==============
// PE suspends its operation and enters a low-power state
// if the Event Register is clear when the WFE is executed

WaitForEvent()
    if EventRegister == '0' then
        EnterLowPowerState();
    return;
```

Library pseudocode for shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()
// ==================
// PE suspends its operation to enter a low-power state
// until a WFI wake-up event occurs or the PE is reset

WaitForInterrupt()
    EnterLowPowerState();
    return;
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictable

```
// ConstrainUnpredictable()
// ========================
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.

Constraint ConstrainUnpredictable(Unpredictable which)
    case which of
        when Unpredictable_WBOVERLAPLD
            return Constraint_WBSUPPRESS; // return loaded value
        when Unpredictable_WBOVERLAPST
            return Constraint_NONE;     // store pre-writeback value
        when Unpredictable_LDPOVERLAP
            return Constraint_UNDEF;    // instruction is UNDEFINED
        when Unpredictable_BASEOVERLAP
            return Constraint_NONE;     // use original address
        when Unpredictable_DATAOVERLAP
            return Constraint_NONE;     // store original value
        when Unpredictable_DEVPAGE2
            return Constraint_FAULT;    // take an alignment fault
        when Unpredictable_INSTRDEVICE
            return Constraint_NONE;     // Do not take a fault
        when Unpredictable_RESCPACR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESMAIR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESTEXCB
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESDACR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESPRRR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESVTCRS
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESTnSZ
            return Constraint_FORCE;    // Map to the limit value
        when Unpredictable_OORTnSZ
            return Constraint_FORCE;    // Map to the limit value
        when Unpredictable_LARGEIPA
            return Constraint_FORCE;    // Restrict the inputsize 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_AFUPDATE     // AF update for alignment or permission fault
            return Constraint_TRUE;
        when Unpredictable_IESBinDebug  // Use SCTLR[].IESB in Debug state
            return Constraint_TRUE;
        when Unpredictable_CLEARERRITEZERO // Clearing sticky errors when instruction in flight
            return Constraint_FALSE;
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

```
// ConstrainUnpredictableBits()
// ============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint_UNKNOWN then
        return (c, Zeros(width));           // See notes; this is an example implementation only
    else
        return (c, bits(width) UNKNOWN);    // bits result not used
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

boolean ConstrainUnpredictableBool(Unpredictable which)

    c = ConstrainUnpredictable(which);
    assert c IN {Constraint_TRUE, Constraint_FALSE};
    return (c == Constraint_TRUE);
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// ===============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the ARMv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the integer part
// of the result.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high, Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint_UNKNOWN then
        return (c, low);                // See notes; this is an example implementation only
    else
        return (c, integer UNKNOWN);    // integer result not used
```

Library pseudocode for shared/functions/unpredictable/Constraint

```
enumeration Constraint    {// General:
                           Constraint_NONE,              // Instruction executes with
                                                         //  no change or side-effect to its described
                           Constraint_UNKNOWN,           // Destination register has UNKNOWN value
                           Constraint_UNDEF,             // Instruction is UNDEFINED
                           Constraint_UNDEFEL0,          // Instruction is UNDEFINED at EL0 only
                           Constraint_NOP,               // Instruction executes as NOP
                           Constraint_TRUE,
                           Constraint_FALSE,
                           Constraint_DISABLED,
                           Constraint_UNCOND,            // Instruction executes unconditionally
                           Constraint_COND,              // Instruction executes conditionally
                           Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
                           // Load-store:
                           Constraint_WBSUPPRESS, Constraint_FAULT,
                           // IPA too large
                           Constraint_FORCE, Constraint_FORCENOSLCHECK};
```

Library pseudocode for shared/functions/unpredictable/Unpredictable

```
enumeration Unpredictable {// Writeback/transfer register overlap (load):
                           Unpredictable_WBOVERLAPLD,
                           // Writeback/transfer register overlap (store):
                           Unpredictable_WBOVERLAPST,
                           // Load Pair transfer register overlap:
                           Unpredictable_LDPOVERLAP,
                           // Store-exclusive base/status register overlap
                           Unpredictable_BASEOVERLAP,
                           // Store-exclusive data/status register overlap
                           Unpredictable_DATAOVERLAP,
                           // Load-store alignment checks:
                           Unpredictable_DEVPAGE2,
                           // Instruction fetch from Device memory
                           Unpredictable_INSTRDEVICE,
                           // Reserved CPACR value
                           Unpredictable_RESCPACR,
                           // Reserved MAIR value
                           Unpredictable_RESMAIR,
                           // Reserved TEX:C:B value
                           Unpredictable_RESTEXCB,
                           // Reserved PRRR value
                           Unpredictable_RESPRRR,
                           // Reserved DACR field
                           Unpredictable_RESDACR,
                           // Reserved VTCR.S value
                           Unpredictable_RESVTCRS,
                           // Reserved TCR.TnSZ value
                           Unpredictable_RESTnSZ,
                           // Out-of-range TCR.TnSZ value
                           Unpredictable_OORTnSZ,
                           // IPA size exceeds PA size
                           Unpredictable_LARGEIPA,
                           // Syndrome for a known-passing conditional A32 instruction
                           Unpredictable_ESRCONDPASS,
                           // Illegal State exception: zero PSTATE.IT
                           Unpredictable_ILZEROIT,
                           // Illegal State exception: zero PSTATE.T
                           Unpredictable_ILZEROT,
                           // Debug: prioritization of Vector Catch
                           Unpredictable_BPVECTORCATCHPRI,
                           // Debug Vector Catch: match on 2nd halfword
                           Unpredictable_VCMATCHHALF,
                           // Debug Vector Catch: match on Data Abort or Prefetch abort
                           Unpredictable_VCMATCHDAPA,
                           // Debug watchpoints: non-zero MASK and non-ones BAS
                           Unpredictable_WPMASKANDBAS,
                           // Debug watchpoints: non-contiguous BAS
                           Unpredictable_WPBASCONTIGUOUS,
                           // Debug watchpoints: reserved MASK
                           Unpredictable_RESWPMASK,
                           // Debug watchpoints: non-zero MASKed bits of address
                           Unpredictable_WPMASKEDBITS,
                           // Debug breakpoints and watchpoints: reserved control bits
                           Unpredictable_RESBPWPCTRL,
                           // Debug breakpoints: not implemented
                           Unpredictable_BPNOTIMPL,
                           // Debug breakpoints: reserved type
                           Unpredictable_RESBPTYPE,
                           // Debug breakpoints: not-context-aware breakpoint
                           Unpredictable_BPNOTCTXCMP,
                           // Debug breakpoints: match on 2nd halfword of instruction
                           Unpredictable_BPMATCHHALF,
                           // Debug breakpoints: mismatch on 2nd halfword of instruction
                           Unpredictable_BPMISMATCHHALF,
                           // Debug: restart to a misaligned AArch32 PC value
                           Unpredictable_RESTARTALIGNPC,
                           // Debug: restart to a not-zero-extended AArch32 PC value
                           Unpredictable_RESTARTZEROUPPERPC,
                           // Zero top 32 bits of X registers in AArch32 state:
                           Unpredictable_ZEROUPPER,
```

```
                                // Zero top 32 bits of PC on illegal return to AArch32 state
                                Unpredictable_ERETZEROUPPERPC,
                                // Force address to be aligned when interworking branch to A32 state
                                Unpredictable_A32FORCEALIGNPC,
                                // SMC disabled
                                Unpredictable_SMD,
                                // Access Flag Update by HW
                                Unpredictable_AFUPDATE,
                                // Consider SCTLR[].IESB in Debug state
                                Unpredictable_IESBinDebug,
                                // No events selected in PMSEVFR_EL1
                                Unpredictable_ZEROPMSEVFR,
                                // No operation type selected in PMSFCR_EL1
                                Unpredictable_NOOPTYPES,
                                // Zero latency in PMSLATFR_EL1
                                Unpredictable_ZEROMINLATENCY,
                                // Clearing DCC/ITR sticky flags when instruction is in flight
                                Unpredictable_CLEARERRITEZERO,
                                // To be determined -- THESE SHOULD BE RESOLVED (catch-all)
                                Unpredictable_TBD};
```

Library pseudocode for shared/functions/vector/AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);

    return imm64;
```

Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

Library pseudocode for shared/functions/vector/SatQ

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```

Library pseudocode for shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N IN {16,32};
    if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)

        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        case N of
            when 16 estimate = RecipSqrtEstimate(UInt(operand<15:7>));
            when 32 estimate = RecipSqrtEstimate(UInt(operand<31:23>));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N IN {16,32};
    if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        case N of
            when 16 estimate = RecipEstimate(UInt(operand<15:7>));
            when 32 estimate = RecipEstimate(UInt(operand<31:23>));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

Library pseudocode for shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

Library pseudocode for shared/translation/attrs/CombineS1S2AttrHints

```
// CombineS1S2AttrHints()
// ======================

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc)

    MemAttrHints result;

    if s2desc.attrs == '01' || s1desc.attrs == '01' then
        result.attrs = bits(2) UNKNOWN;   // Reserved
    elsif s2desc.attrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
        result.attrs = MemAttr_NC;        // Non-cacheable
    elsif s2desc.attrs == MemAttr_WT || s1desc.attrs == MemAttr_WT then
        result.attrs = MemAttr_WT;        // Write-through
    else
        result.attrs = MemAttr_WB;        // Write-back

    result.hints = s1desc.hints;
    result.transient = s1desc.transient;

    return result;
```

Library pseudocode for shared/translation/attrs/CombineS1S2Desc

```
// CombineS1S2Desc()
// =================
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    AddressDescriptor result;

    result.paddress = s2desc.paddress;

    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    elsif s2desc.memattrs.type == MemType_Device || s1desc.memattrs.type == MemType_Device then
        result.memattrs.type = MemType_Device;
        if s1desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s2desc.memattrs.device;
        elsif s2desc.memattrs.type == MemType_Normal then
            result.memattrs.device = s1desc.memattrs.device;
        else                      // Both Device
            result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
                                                       s2desc.memattrs.device);
    else                          // Both Normal
        result.memattrs.type = MemType_Normal;
        result.memattrs.device = DeviceType UNKNOWN;
        result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner);
        result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer);
        result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
        result.memattrs.outershareable = (s1desc.memattrs.outershareable ||
                                          s2desc.memattrs.outershareable);

    result.memattrs = MemAttrDefaults(result.memattrs);

    return result;
```

Library pseudocode for shared/translation/attrs/CombineS1S2Device

```
// CombineS1S2Device()
// ===================
// Combines device types from stage 1 and stage 2

DeviceType CombineS1S2Device(DeviceType s1device, DeviceType s2device)

    if s2device == DeviceType_nGnRnE || s1device == DeviceType_nGnRnE then
        result = DeviceType_nGnRnE;
    elsif s2device == DeviceType_nGnRE || s1device == DeviceType_nGnRE then
        result = DeviceType_nGnRE;
    elsif s2device == DeviceType_nGRE || s1device == DeviceType_nGRE then
        result = DeviceType_nGRE;
    else
        result = DeviceType_GRE;

    return result;
```

Library pseudocode for shared/translation/attrs/LongConvertAttrsHints

```
// LongConvertAttrsHints()
// =======================
// Convert the long attribute fields for Normal memory as used in the MAIR fields
// to orthogonal attributes and hints

MemAttrHints LongConvertAttrsHints(bits(4) attrfield, AccType acctype)
    assert !IsZero(attrfield);
    MemAttrHints result;
    if S1CacheDisabled(acctype) then             // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        if attrfield<3:2> == '00' then          // Write-through transient
            result.attrs = MemAttr_WT;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        elsif attrfield<3:0> == '0100' then     // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
            result.transient = FALSE;
        elsif attrfield<3:2> == '01' then       // Write-back transient
            result.attrs = attrfield<1:0>;
            result.hints = MemAttr_WB;
            result.transient = TRUE;
        else                                    // Write-through/Write-back non-transient
            result.attrs = attrfield<3:2>;
            result.hints = attrfield<1:0>;
            result.transient = FALSE;

    return result;
```

Library pseudocode for shared/translation/attrs/MemAttrDefaults

```
// MemAttrDefaults()
// =================
// Supply default values for memory attributes, including overriding the shareability attributes
// for Device and Non-cacheable memory types.

MemoryAttributes MemAttrDefaults(MemoryAttributes memattrs)

    if memattrs.type == MemType_Device then
        memattrs.inner = MemAttrHints UNKNOWN;
        memattrs.outer = MemAttrHints UNKNOWN;
        memattrs.shareable = TRUE;
        memattrs.outershareable = TRUE;
    else
        memattrs.device = DeviceType UNKNOWN;
        if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
            memattrs.shareable = TRUE;
            memattrs.outershareable = TRUE;

    return memattrs;
```

Library pseudocode for shared/translation/attrs/S1CacheDisabled

```
// S1CacheDisabled()
// =================

boolean S1CacheDisabled(AccType acctype)
    if ELUsingAArch32(S1TranslationRegime()) then
        if PSTATE.EL == EL2 then
            enable = if acctype == AccType_IFETCH then HSCTLR.I else HSCTLR.C;
        else
            enable = if acctype == AccType_IFETCH then SCTLR.I else SCTLR.C;
    else
        enable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;
    return enable == '0';
```

Library pseudocode for shared/translation/attrs/S2AttrDecode

```
// S2AttrDecode()
// ==============
// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)

    MemoryAttributes memattrs;

    if attr<3:2> == '00' then                    // Device
        memattrs.type = MemType_Device;
        case attr<1:0> of
            when '00'  memattrs.device = DeviceType_nGnRnE;
            when '01'  memattrs.device = DeviceType_nGnRE;
            when '10'  memattrs.device = DeviceType_nGRE;
            when '11'  memattrs.device = DeviceType_GRE;

    elsif attr<1:0> != '00' then                // Normal
        memattrs.type = MemType_Normal;
        memattrs.outer = S2ConvertAttrsHints(attr<3:2>, acctype);
        memattrs.inner = S2ConvertAttrsHints(attr<1:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';

    else
        memattrs = MemoryAttributes UNKNOWN;    // Reserved

    return MemAttrDefaults(memattrs);
```

Library pseudocode for shared/translation/attrs/S2CacheDisabled

```
// S2CacheDisabled()
// =================

boolean S2CacheDisabled(AccType acctype)
    if ELUsingAArch32(EL2) then
        disable = if acctype == AccType_IFETCH then HCR2.ID else HCR2.CD;
    else
        disable = if acctype == AccType_IFETCH then HCR_EL2.ID else HCR_EL2.CD;

    return disable == '1';
```

Library pseudocode for shared/translation/attrs/S2ConvertAttrsHints

```
// S2ConvertAttrsHints()
// =====================
// Converts the attribute fields for Normal memory as used in stage 2
// descriptors to orthogonal attributes and hints

MemAttrHints S2ConvertAttrsHints(bits(2) attr, AccType acctype)
    assert !IsZero(attr);

    MemAttrHints result;

    if S2CacheDisabled(acctype) then   // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case attr of
            when '01'                   // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '10'                   // Write-through
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RWA;
            when '11'                   // Write-back
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;

    result.transient = FALSE;

    return result;
```

Library pseudocode for shared/translation/attrs/ShortConvertAttrsHints

```
// ShortConvertAttrsHints()
// ========================
// Converts the short attribute fields for Normal memory as used in the TTBR and
// TEX fields to orthogonal attributes and hints

MemAttrHints ShortConvertAttrsHints(bits(2) RGN, AccType acctype, boolean secondstage)

    MemAttrHints result;

    if (!secondstage && S1CacheDisabled(acctype)) || (secondstage && S2CacheDisabled(acctype)) then
       // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case RGN of
            when '00'                   // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '01'                   // Write-back, Read and Write allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;
            when '10'                   // Write-through, Read allocate
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RA;
            when '11'                   // Write-back, Read allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RA;

    result.transient = FALSE;

    return result;
```

Library pseudocode for shared/translation/attrs/WalkAttrDecode

```
// WalkAttrDecode()
// ================

MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN, boolean secondstage)

    MemoryAttributes mematrs;

    AccType acctype = AccType_NORMAL;

    mematrs.type = MemType_Normal;
    mematrs.inner = ShortConvertAttrsHints(IRGN, acctype, secondstage);
    mematrs.outer = ShortConvertAttrsHints(ORGN, acctype, secondstage);
    mematrs.shareable = SH<1> == '1';
    mematrs.outershareable = SH == '10';

    return MemAttrDefaults(mematrs);
```

Library pseudocode for shared/translation/translation/HasS2Translation

```
// HasS2Translation()
// ==================
// Returns TRUE if stage 2 translation is present for the current translation regime

boolean HasS2Translation()
    return (HaveEL(EL2) && !IsSecure() && !IsInHost() && PSTATE.EL IN {EL0,EL1});
```

Library pseudocode for shared/translation/translation/Have16bitVMID

```
// Have16bitVMID()
// ===============
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return HaveEL(EL2) && boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/translation/translation/IsEL1TransRegimeRegs

```
// IsEL1TransRegimeRegs()
// ======================

// Returns TRUE if its a register in EL1 translation regime

boolean IsEL1TransRegimeRegs()
    return !HaveEL(EL2) || PSTATE.EL == EL1 || (PSTATE.EL == EL0 && (HCR_EL2.E2H == '0' ||  HCR_EL2.TGE
```

Library pseudocode for shared/translation/translation/PAMax

```
// PAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the physical address
// size for this processor, as log2().

integer PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

Library pseudocode for shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

Library pseudocode for shared/translation/translation/VAMax

```
// VAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the virtual address
// size for this processor, as log2().

integer VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
```

Internal version only: isa v25.11, AdvSIMD v25.02, pseudocode v32.2

Copyright © 2010-2017 ARM Limited or its affiliates. All rights reserved. This document is Non-Confidential.